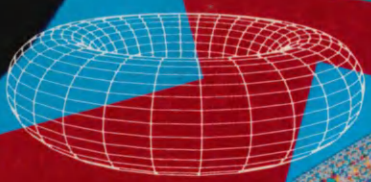
WPC
WILEY
WILEY PROFESSIONAL COMPUTING

PROGRAMMING PRINCIPLES IN COMPUTER GRAPHICS Second Edition
Ammeraal

PROGRAMMING PRINCIPLES IN COMPUTER GRAPHICS

Second Edition



Leendert Ammeraal

DISK AVAILABLE

WILEY

AVAILABLE

The programs described in this book are available on 3½″ disk* for your IBM PC (and most compatibles). They can be compiled by the Turbo C++ and Borland C++ compilers. Your computer will also need a VGA graphics adaptor.

Order the Program Disk today, priced £15.00 (includes VAT)/$23.00 from your computer store, bookseller, or by using the order form below. (Prices correct at time of going to press.)

*5¼″ disks are available on request

Ammeraal: Programming Principles in Computer Graphics, Second Edition — Program Disk

Please send me ..... copies of the **Ammeraal: Programming Principles in Computer Graphics, Seoond Edition — Program Disk** at £15.00 (includes VAT)/$23.00 each.

0 471 93129 2

POSTAGE AND HANDLING FREE FOR CASH WITH ORDER OR PAYMENT BY CREDIT CARD

☐ Remittance enclosed ..... Allow approx. 14 days for delivery

☐ Please charge this order to my credit card (All orders subject to credit approval)

Delete as necessary:—AMERICAN EXPRESS, DINERS CLUB, BARCLAYCARD/VISA, ACCESS/MASTERCARD

CARD NUMBER ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiry date .....

☐ Please send me an invoice for prepayment. A small postage and handling charge will be made.

Software purchased for professional purposes is generally recognized as tax deductible.

NAME/ADDRESS .....

.....

.....

OFFICIAL ORDER NUMBER ..... SIGNATURE .....

If you have any queries please contact:

Helen Ramsey
John Wiley & Sons Limited
Baffins Lane
Chichester
West Sussex
PO19 1UD
England

Affix
stamp
here

Customer Service Department
John Wiley & Sons Limited
Shripney Road
Bognor Regis
West Sussex
PO22 9SA
England

PROGRAMMING PRINCIPLES IN COMPUTER GRAPHICS *Second Edition*

PROGRAMMING PRINCIPLES IN COMPUTER GRAPHICS *Second Edition*

Leendert Ammeraal
Hogeschool Utrecht, The Netherlands

JOHN WILEY & SONS
Chichester · New York · Brisbane · Toronto · Singapore

Other Wiley Editorial Offices

John Wiley & Sons, Inc., 605 Third Avenue,
New York, NY 10158-0012, USA

Jacaranda Wiley Ltd, G.P.O. Box 859, Brisbane,
Queensland 4001, Australia

John Wiley & Sons (Canada) Ltd, 22 Worcester Road,
Rexdale, Ontario M9W 1L1, Canada

John Wiley & Sons (SEA) Pte Ltd, 37 Jalan Pemimpin #05-04,
Block B, Union Industrial Building, Singapore 2057

British Library Cataloguing in Publication Data

A catalogue record for this book is available
from the British Library

ISBN 0 471 93128 4

Printed and bound in Great Britain by Courier International Ltd, East Kilbride

Contents

Preface

This book is about mathematical and programming aspects of computer graphics. It is primarily intended for those who want to write graphics programs themselves and, in contrast to most other books on computer graphics, it is based on the C++ language. A sympathetic aspect of C++ is that it has so many good programming facilities in common with C; in other words, when programming in C++ we can deviate from C as much or as little as we like. The more spectacular new aspects of C++ are not always used in this book. Being unfamiliar with C++ will therefore not be a serious handicap, provided you can read C programs.

Besides switching from C to C++, there are some other new points in this edition. The important Bresenham algorithms for line and circle drawing are now included, and so is the subject of polygon filling. This is related to hidden-face elimination, which is also new in this edition. Emphasis is now more on *reusable program modules*. Most programs consist of an application module and several implementation modules, with corresponding header files as interfaces.

The file format for 3D objects used in this book is the same as that in *Interactive 3D Computer Graphics*, which is mainly about one program, D3D. The present book, on the other hand, is primarily intended as a textbook and therefore more general in two respects: it deals also with 2D graphics and, except for Appendix C, it is machine independent. Its first edition did very well in the first two or three years after its publication, but it badly needed a revision because all programs in it were in the old C style. Instead of only switching to the new style, recommended by ANSI and mandatory in C++, it seemed a good idea to me to add also some new subjects, such as those mentioned. This second edition will reduce the amount of paperwork involved in handing out supplementary lecture notes to my students. I hope that this will also apply to some colleagues. Any comments would be very welcome.

Leendert Ammeraal

1

Introduction

1.1 Graphics Programming and the C++ Language

It is hardly possible to find a subject that is more controversial than programming languages. Let us therefore consider some facts, rather than opinions. A discussion about programming languages used in computer graphics would be incomplete without mentioning Fortran, which was the dominant language for professionals for a very long time. In the eighties, many programmers switched from Fortran to the C language, which has many advantages, such as, for example, the possibility of recursion. About 1990, the C++ language became popular because of its facilities for object-oriented programming (OOP). Unlike some other OOP languages, C++ can also be used as a conventional language, that is, as 'a better C'. This is also the case with ANSI C, but this is still very tolerant with regard to old-style programming practice, such as calling functions that are not declared previously. By contrast, C++ requires the new style; it accepts calls to a function only if complete information about the parameters of that function is available. Now that very good C++ compilers are available, especially under Unix and on the IBM PC, it is to be expected that C++ will soon be widely regarded as the successor to C. Some convenient new aspects are

(1) operator definitions for user-defined types,
(2) constructors used to create objects of user-defined type,
(3) type-safe linkage.

Applying (1) and (2) can be regarded as extending the language to a new, more problem-oriented one. New types and operators are declared in header files, and they are implemented in separate modules. For example, using a properly defined vector type **vec**, we can write

```
vec v(2, 3), s;
s = v + vec(1, 0);  // s is equal to vec(3, 4)
```

to compute the sum of vector v and the unit vector (1, 0). This example illustrates the points (1) and (2): it shows that our own plus-operator can be applied to vectors and that a vector object can be easily created when its *x* and *y* components are given.

This example also shows a potential weakness of this approach. Computing the sum s of the two vectors v and (1, 0) will be more efficient if we write

```
s.x = v.x + 1;
s.y = v.y;
```

Not only does this prevent computing the sum v.y + 0, but it also avoids calling the *constructor* **vec** and copying 1 and 0 into a newly created **vec** object. We will therefore often use the more traditional (ANSI) C programming style.

Nevertheless, we will often benefit from the fact that we are using C++. For example, the good practice of declaring functions before they are used is (only) *recommended* in ANSI C, while it is *mandatory* in C++. As in C programs, such declarations of functions defined elsewhere are normally placed in header files, so that consistency is guaranteed. But even if we are inconsistent in this regard, we will obtain a linker error in C++ but not in C. For example, there is such an error if we declare and use function **f** as **void f(void)** in one module and define it as **void f(int)** in another. An error message from the linker is possible in C++ because of *type-safe linkage*, mentioned in point (3), which means that the linker is supplied with full information about parameter types.

1.2 Our First Graphics Programs

We will not discuss the C++ language here in a systematic way, so a book[1] on this language may be required if you are not yet familiar with it. On the other hand, many programs will be rather simple, so you may understand their meanings even if they contain some language constructs that are new to you. Here is our first C++ program. Except for the way comments are written, it is at the same time a C program:

```
// SQUARES: This program draws 50 squares inside
//          each other. To be linked with module GRSYS.
#include "grsys.h"

int main()
{  float xA, yA, xB, yB, xC, yC, xD, yD,
         xA1, yA1, xB1, yB1, xC1, yC1, xD1, yD1, p, q, r;
   int i;
```

[1] See *C++ for Programmers*, by the same author and from the same publisher as this book.

```
    q = 0.05; p = 1 - q; // q = lambda (see discussion below)
    initgr();
    r = 0.95 * r_max;
    xA = xD = x_center - r;
    xB = xC = x_center + r;
    yA = yB = y_center - r;
    yC = yD = y_center + r;
    for (i=0; i<50; i++)
    {  move(xA, yA);
       draw(xB, yB); draw(xC, yC);
       draw(xD, yD); draw(xA, yA);
       xA1=p*xA+q*xB; yA1=p*yA+q*yB;
       xB1=p*xB+q*xC; yB1=p*yB+q*yC;
       xC1=p*xC+q*xD; yC1=p*yC+q*yD;
       xD1=p*xD+q*xA; yD1=p*yD+q*yA;
       xA=xA1; xB=xB1; xC=xC1; xD=xD1;
       yA=yA1; yB=yB1; yC=yC1; yD=yD1;
    }
    endgr();
    return 0;
}
```

The output of this program is shown in Fig. 1.1. There are calls to four graphics functions, declared in the header file GRSYS.H:

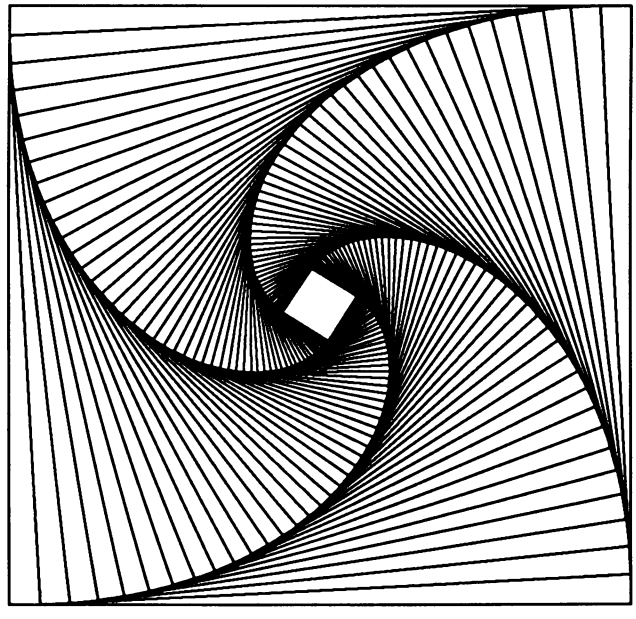


Fig. 1.1. Output of program SQUARES

initgr( )	initializes graphic output;
move(x, y)	moves a (real of fictitious) pen to point (x, y)[1];
draw(x, y)	draws a line segment from the current pen position to point (x, y);
endgr( )	performs any final actions (such as switching back from graphics mode to text mode) after a key has been pressed.

The following external variables are also declared in GRSYS.H; since function **initgr** assigns appropriate values to them, they should be used only after a call to that function. The first three of these seven variables are used in program SQUARES:

x_center, *y_center*	coordinates of screen center;
r_max	radius of largest possible circle on the screen;
x_min, *y_min*	coordinates of lower-left corner of the screen;
x_max, *y_max*	coordinates of upper-right corner of the screen.

A call to **initgr** is required before calls to the functions **move** and **draw**. Similarly, there must be a call to **endgr** after the final call to **move** or **draw**. The two calls **move(x, y)** and **draw(x, y)** have in common that they move a (real or fictitious) pen to point (x, y); with **move(x, y)** this pen is up and with **draw(x, y)** it is down.

The four functions just mentioned do not belong to the C++ language. They are external routines; after compilation of our program they are added to it by the linker. The definitions of the functions and variables mentioned above occur in a separate module, GRSYS.CPP, which is system dependent and therefore not included in this chapter. However, a version of this module for the IBM PC (and compatible machines) is included in Appendix C. This is intended to be the only system-dependent element in this book: all other programs and program modules should be accepted by any C++ compiler. For other computer systems, a modified version of GRSYS.CPP will be required. To make this as easy as possible, this module will be kept limited in size. Some other useful graphics functions will therefore be located in other modules, which are device-independent.

'Program' SQUARES is in fact not a complete program but rather a program *module*. After compiling this file (with the full name SQUARES.CPP), its resulting *object module* SQUARES.OBJ is to be linked together with GRSYS.OBJ, which was produced earlier by compiling GRSYS.CPP.

The declarations of the above functions and variables can be found in the header file GRSYS.H. Because of the quotation marks in the line

```
#include "grsys.h"
```

which occurs in program SQUARES, this header file must be present in the current directory when our program is compiled:

---

[1] Program fragments are normally printed in boldface in this book. However, program variables denoting numbers and standing on their own are printed in italics because they frequently also occur in mathematical expressions, in which boldface is used for vectors.

```
// GRSYS.H: Graphics primitives
extern float x_min, x_max, y_min, y_max, x_center, y_center,
             r_max;
void initgr(char *hpgfile=0);
void endgr(void);
void move(float x, float y);
void draw(float x, float y);
...
```

As you can see, function **initgr** has a default parameter **hpgfile**. We can use this to obtain our graphics output in a file, which can subsequently be imported by many text processors and desktop publishing packages. A suitable format for this file (used in the version of GRSYS.CPP listed in Appendix C) is HP-GL, originally designed for pen plotters but now also widely in use for other purposes. HP-GL files are *vector oriented*, which means that each line segment is identified by its two end points. Consequently, the quality of the final result on paper depends only on the printer or plotter that we are using, and not on the video display. We could instead have used a screen-capture utility (such as GRAB from WordPerfect), but then the file would be *bit oriented*, with a resolution based on the video display and resulting in a lower quality of the hard copy that we will eventually produce. If we want a file, say, SQUARES.HPG, of our set or squares, we can write

```
initgr("squares.hpg");
```

instead of calling **initgr** without arguments, as is done in program SQUARES. The graphics output shown in Fig. 1.1 consists of fifty squares.

A square ABCD is drawn and then a new point A′ is chosen on side AB such that AA′ = 0.05 × AB. As Fig. 1.2 shows, we can associate the points A, B and A′ with the vectors **a** = **OA**, **b** = **OB** and **a′** = **OA′**.

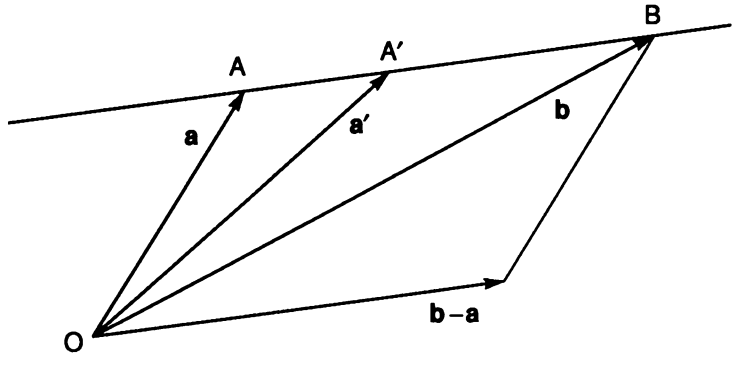


Fig. 1.2. Points and vectors

Then for any point A′ on the line AB the following vector equation applies:

$$\mathbf{a}' = \mathbf{a} + \lambda(\mathbf{b} - \mathbf{a})$$

which we can also write as

$$\mathbf{a}' = (1 - \lambda)\mathbf{a} + \lambda\mathbf{b}$$

In terms of coordinates, this is written as

$$x_{A'} = (1 - \lambda)x_A + \lambda x_B$$
$$y_{A'} = (1 - \lambda)y_A + \lambda y_B$$

Point A′ coincides with A if $\lambda = 0$, and with B if $\lambda = 1$. The value $\lambda = 0.05$ was used in program SQUARES to make point A′ lie near A on line segment AB. Points B′, C′ and D′ are chosen similarly on the sides BC, CD and DA, respectively. The procedure is then repeated with A′, B′, C′ and D′ as the new points A, B, C and D, respectively.

A generalized program for squares

Program SQUARES does not read any input data and can produce only one picture. It is normally desirable for graphics programs to be more general, so that we can use

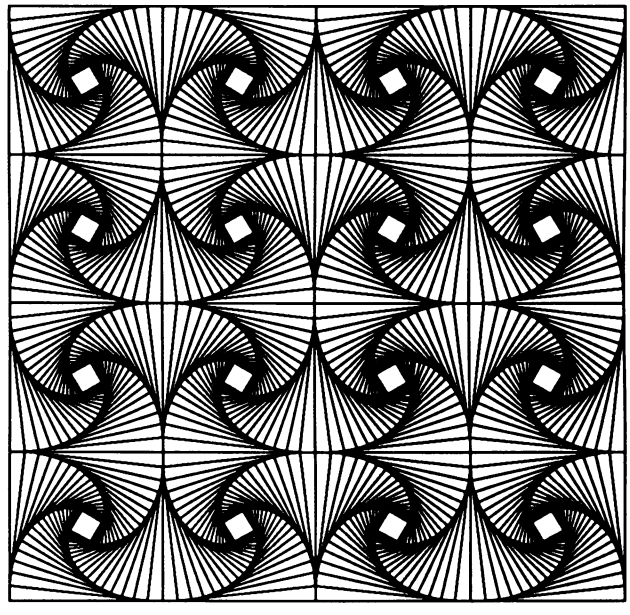


Fig. 1.3. Sample output of program MANYSQ

input data to specify details about what we want. In our example, we may as well draw more than one set of squares, say, $n \times n$ of them, arranged in a larger square, like the squares of a chessboard. We also want to use a variable number m of squares in each set, rather than exactly 50. Finally, we want to supply λ as input data, instead of always using $\lambda = 0.05$. Program MANYSQ is such a generalized version. Figure 1.3 shows an example of its output. It was obtained by using the input data $n = 4$, $m = 20$, and $\lambda = 0.9$. Actually, the $n \times n$ squares are divided into two types, like the black and white squares of a chessboard, by alternate use of the values λ and $1 - \lambda$ for p (each time q being equal to $1 - p$).

If you are not yet familiar with C++, you may also find program MANYSQ instructive in that it shows how to display text on the screen and to read input data from the keyboard:

```
/* MANYSQ: This program draws n x n sets of squares, arranged
           as on a chessboard. To be linked with module GRSYS.
*/
#include <iostream.h>
#include "grsys.h"

void set_of_squares(float xA, float yA, int m,
                    float p, float a)
{  float xB=xA+a, yB=yA, xC=xB, yC=yA+a, xD=xA, yD=yC,
   xA1, yA1, xB1, yB1, xC1, yC1, xD1, yD1, q=1-p;
   int i;
   for (i=0; i<m; i++)
   {  move(xA, yA);
      draw(xB, yB); draw(xC, yC);
      draw(xD, yD); draw(xA, yA);
      xA1=p*xA+q*xB; yA1=p*yA+q*yB;
      xB1=p*xB+q*xC; yB1=p*yB+q*yC;
      xC1=p*xC+q*xD; yC1=p*yC+q*yD;
      xD1=p*xD+q*xA; yD1=p*yD+q*yA;
      xA=xA1; xB=xB1; xC=xC1; xD=xD1;
      yA=yA1; yB=yB1; yC=yC1; yD=yD1;
   }
}

int main()
{  int m, n, i, j;
   float a, lambda, halfn;
   cout << "There will be n x n sets of squares.\n";
   cout << "Enter n (e.g. 8 for a chessboard): ";
   cin >> n; halfn = 0.5 * n;
   cout << "How many squares in each set? (e.g. 10): ";
   cin >> m;
   cout << "Enter interpolation factor between 0 and 1" ;
```

```
    cout << "(e.g. 0.2): "; cin >> lambda;
    initgr("manysq.hpg");
    a = 1.9 * r_max/n; // a = length of side of largest square
    for (i=0; i<n; i++)
    for (j=0; j<n; j++)
       set_of_squares(x_center + (i - halfn) * a,
                      y_center + (j - halfn) * a,
                      m,
                      ((i + j) % 2 ? lambda : 1 - lambda),
                      a);
    endgr(); return 0;
}
```

Exercises

1.1 Write program TRIANGLES to draw a set of triangles inside each other, similar to the way the squares are drawn in program SQUARES. The coordinates of vertices A, B, and C of the outermost triangle are to be read from the keyboard.

1.2 The equations

$$x = x_C + r \cos \varphi$$
$$y = y_C + r \sin \varphi$$

form a parameter representation of the circle with center $C(x_C, y_C)$ and radius r. Parameter φ ranges from 0 to 2π. We can use this to draw a regular polygon that approximates a circle. Using, for example, $n = 60$, we compute

$$\theta = 2\pi/n$$

and use the values $\varphi = i \cdot \theta$, where $i = 0, 1, \ldots, n-1$. In this way we find n points on a circle. If we connect all neighboring pairs of points found in this way then, with the large value of n mentioned, the resulting regular polygon will be a good approximation of a circle. Write a program which uses this method to draw 20 concentric circles.

1.3 Write a program that draws a chessboard as shown in Fig. 1.4, with hatching used for the squares that are normally black. An integer m is to be read from the keyboard. It is the number of intervals into which the sides of the 'black' squares are to divided. In Fig. 1.4 we have $m = 5$.

1.4 Write a program that draws a pattern of hexagons, as shown in Fig. 1.5. The vertices of a (regular) hexagon lie on its so-called circumscribed circle. Your program must read the radius of these circles and draw as many hexagons on the screen as is possible. In Fig. 1.5, the screen boundaries are also shown. Note that the margin on the left is equal to that on the right, and the same

applies to the margins at the top and at the bottom. As follows from our discussion in Section 1.2, you can draw the screen boundaries by using the global variables *x_min*, *x_max*, *y_min*, *y_max*, declared in GRSYS.H.

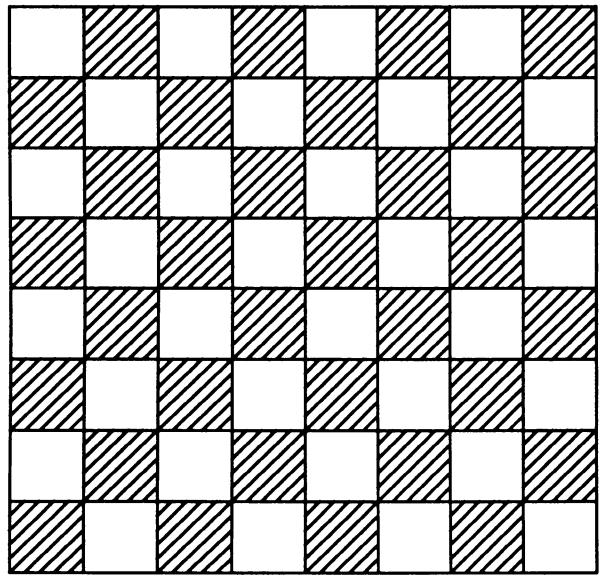


Fig. 1.4. Chessboard

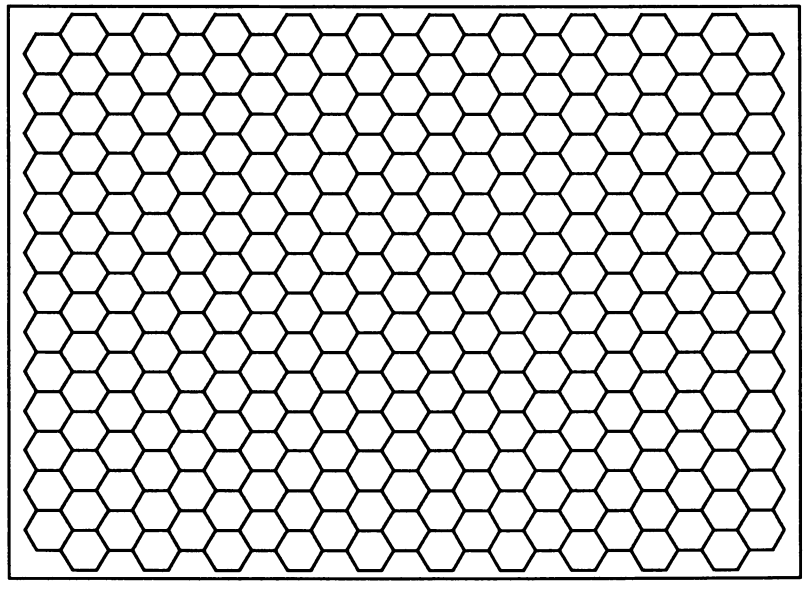


Fig. 1.5. Pattern of hexagons

2

Transformations, Windows and Viewports

2.1 Translations and Rotations

Consider the following system of equations:

$$\begin{cases} x' = x + a \\ y' = y + b \end{cases}$$

We can interpret this in two ways:

(1) As a translation: all points move a units to the right and b units upwards (see Fig. 2.1(a)).
(2) As a change of coordinates: the x- and y-axes move a units to the left and b units downwards (see Fig. 2.1(b)).

This simple example shows a principle which also applies to more complex situations. We will often deal with systems of equations, usually written as matrix products, and interpret them as a transformation of all points in a fixed coordinate system. However, the same system of equations can then be interpreted as a change of coordinates.

We will now rotate point $P(x, y)$ about the origin O through a given angle φ. The image point will be called $P'(x', y')$ (see Fig. 2.2). Then there are numbers a, b, c, d such that x' and y' can be derived from x and y as follows:

$$\begin{cases} x' = ax + by \\ y' = cx + dy \end{cases} \tag{2.1}$$

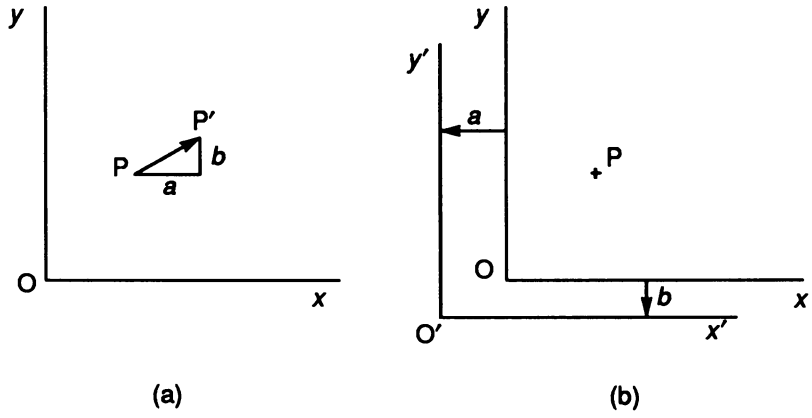


Fig. 2.1. (a) *Translation*; (b) *change of coordinates*

The values a, b, c, d can be obtained by choosing first $(x, y) = (1, 0)$. Setting $x = 1$ and $y = 0$ in Eq. (2.1), we have

$$x' = a$$
$$y' = c$$

However, in this simple case the values of x' and y' are cos φ and sin φ, as Fig. 2.3(a) shows. Thus we have

$$a = \cos \varphi$$
$$c = \sin \varphi$$

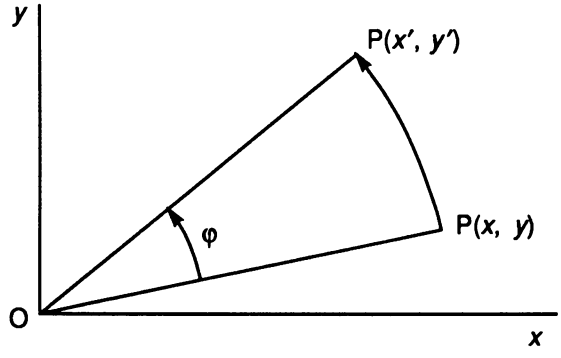


Fig. 2.2. Rotation about O through angle φ

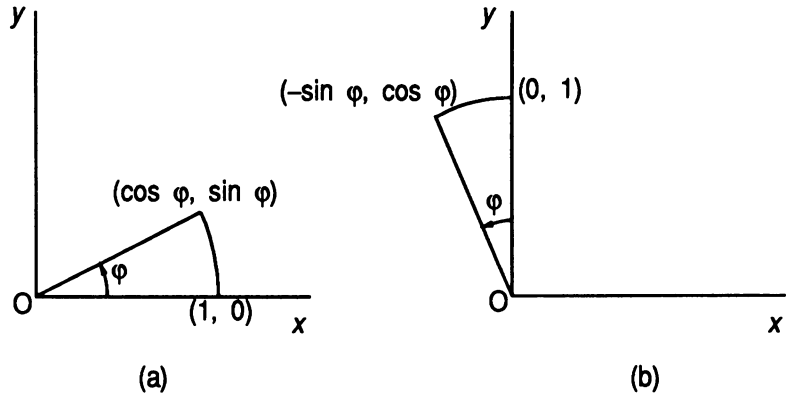


Fig. 2.3. (a) *Image of* (1, 0); (b) *image of* (0, 1)

In the same way, Fig. 2.3(b) shows

$$b = -\sin\varphi$$
$$d = \cos\varphi$$

Thus we now write Eqs. (2.1) as

$$\begin{cases} x' = x\cos\varphi - y\sin\varphi \\ y' = x\sin\varphi + y\cos\varphi \end{cases} \tag{2.2}$$

Equations (2.2) describe a rotation about O. Often this is not what we want. If, instead of point O, we have to use a given point (x_0, y_0) as the center of rotation, we simply replace x with $x - x_0$ and y with $y - y_0$. Analogously, we replace x' and y' with $x' - x_0$ and $y' - y_0$, respectively. Applying these substitutions to (2.2) gives

$$\begin{cases} x' - x_0 = (x - x_0)\cos\varphi - (y - y_0)\sin\varphi \\ y' - y_0 = (x - x_0)\sin\varphi + (y - y_0)\cos\varphi \end{cases}$$

Since we have to compute x' and y', we write this as follows:

$$\begin{cases} x' = x_0 + (x - x_0)\cos\varphi - (y - y_0)\sin\varphi \\ y' = y_0 + (x - x_0)\sin\varphi + (y - y_0)\cos\varphi \end{cases} \tag{2.3}$$

These equations can be used for any rotation in the two-dimensional plane. If objects consisting of line segments are to be rotated, we apply Eqs. (2.3) to the endpoints P and Q of these line segments, which results in the rotated endpoints P′ and Q′. Then P′Q′ is drawn as the rotated image of line segment PQ. We will use this principle in the next section.

2.2 Points and Vectors in C++ Programs

As Fig. 1.2 in Section 1.2 shows, we can associate each point A with the vector that starts at O and ends at A. Similarly, we can associate a given vector **OA** with its endpoint A. Both a point and a vector are identified by a pair of real numbers (x, y). Let us therefore declare type **vec** as a structure with **float** members x and y, and use this type not only (as its name suggests) for *vectors*, but also for *points*. We may as well introduce new versions of the functions **move** and **draw**, each of which takes a point (of type **vec**) as its argument. Remember, the C++ language allows us to use two or more distinct functions with the same name, provided they differ in the numbers or types of their parameters. Both this new type **vec** and these new functions **move** and **draw** are declared in the following graphics header file, VEC.H:

```
// VEC.H: Header file for vector operations

struct vec
{  float x, y;
   vec(float xx, float yy){x = xx; y = yy;}
   vec(){x=0; y=0;}
};

vec operator+(vec &u, vec &v);
vec operator-(vec &u, vec &v);
vec operator*(float c, vec &v);

vec &operator+=(vec &u, vec &v);
vec &operator-=(vec &u, vec &v);
vec &operator*=(vec &v, float c);

void move(vec &P);
void draw(vec &P);
vec rotate(vec &P, vec &C, double cosphi, double sinphi);
```

Since these new functions **move** and **draw** call those of module GRSYS, we must use this module whenever we use module VEC (even if we do not use **move** and **draw** at all). Note the *constructors*, declared within the **vec** structure. They enable us to use type **vec** as is done in Section 1.1. Header file VEC.H also shows the declarations of the new operators +, –, *, +=, –=, and *= for **vec** objects (where * and *= denote multiplication of a vector by a scalar).

Since, unlike GRSYS, module VEC is system independent, we will at the same time consider module VEC.CPP. This includes the function **rotate**, which rotates a point P about a point C through an angle φ. Instead of using this angle as a parameter, we rather use its cosine and sine values: function **rotate** will normally be applied to many points that are to be rotated about the same angle φ. It would then be a waste of computing time if we computed $\cos\varphi$ and $\sin\varphi$ each time this function is called.

```
// VEC.CPP: A module for dealing with vectors
#include "grsys.h"
#include "vec.h"

vec operator+(vec &u, vec &v)
{  return vec(u.x + v.x, u.y + v.y);
}

vec operator-(vec &u, vec &v)
{  return vec(u.x - v.x, u.y - v.y);
}

vec operator*(float c, vec &v)
{  return vec(c * v.x, c * v.y);
}

vec &operator+=(vec &u, vec &v)
{  u.x += v.x; u.y += v.y;
   return u;
}

vec &operator-=(vec &u, vec &v)
{  u.x -= v.x; u.y -= v.y;
   return u;
}

vec &operator*=(vec &v, float c)
{  v.x *= c; v.y *= c;
   return v;
}

vec rotate(vec &P, vec &C, double cosphi, double sinphi)
{  double dx = P.x - C.x, dy = P.y - C.y;
   return vec(C.x + dx * cosphi - dy * sinphi,
              C.y + dx * sinphi + dy * cosphi);
}

void move(vec &P){move(P.x, P.y);}
void draw(vec &P){draw(P.x, P.y);}
```

We will now use this new type vec in a program that rotates an object consisting of several line segments. Figure 2.4 shows an arrow, which we will rotate 30 times about the center of the screen, each time through the angle $\varphi = 360/30 = 12°$. The points P_0, P_1, P_2, and P_3, indicated in Fig. 2.4, are stored in vec array **P**.

The coordinates of these four points are given relative to the center of each arrow as percentages of *r_max* (declared in GRSYS.H and available after a call to **initgr**).

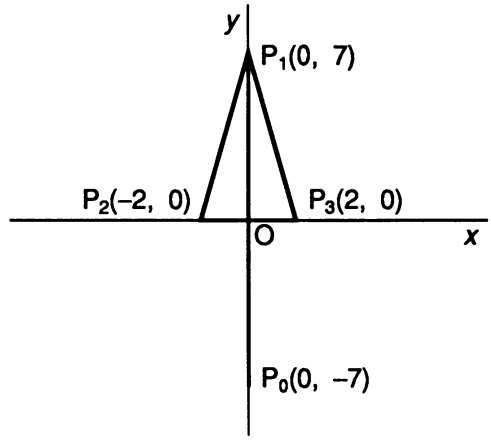


Fig. 2.4. Arrow

```
/* ARROWS30:
   This program draws 30 arrows, flying counter-clockwise
   about the center of the screen.
   It is to be linked with the modules GRSYS and VEC.
*/
#include <math.h>
#include "grsys.h"
#include "vec.h"
double cosphi, sinphi;

int main()
{  int i, j;
   float pi, phi, cosphi, sinphi;
   vec center;
   vec P[4] = {vec(0, -7), vec(0, 7), vec(-2, 0), vec(2, 0)};
   pi = 4 * atan(1.0);
   phi = pi/15;
   cosphi = cos(phi); sinphi = sin(phi);
   initgr();
   center = vec(x_center, y_center);
   // Apply scaling, and move to start position:
   vec startposition = center + vec(0.9 * r_max, 0);
   for (j=0; j<4; j++)
      P[j] = 0.01 * r_max * P[j] + startposition;
   for (i=0; i<30; i++)
   {  // Rotate the arrow:
      for (j=0; j<4; j++)
         P[j] = rotate(P[j], center, cosphi, sinphi);
```

```
        // Draw the rotated arrow:
        move(P[0]); draw(P[1]); draw(P[2]); draw(P[3]);
        draw(P[1]);
    }
    endgr(); return 0;
}
```

After the call to **initgr**, vector addition and scalar multiplication of vectors are used in the program fragment

```
vec startposition = center + vec(0.9 * r_max, 0);
for (j=0; j<4; j++)
    P[j] = r_max * P[j] + startposition;
```

Since the update operators += and *= are also available for vectors, the last program line might be replaced with

```
{   P[j] *= r_max;
    P[j] += startposition;
}
```

The output of program ARROWS30 is shown in Fig. 2.5.

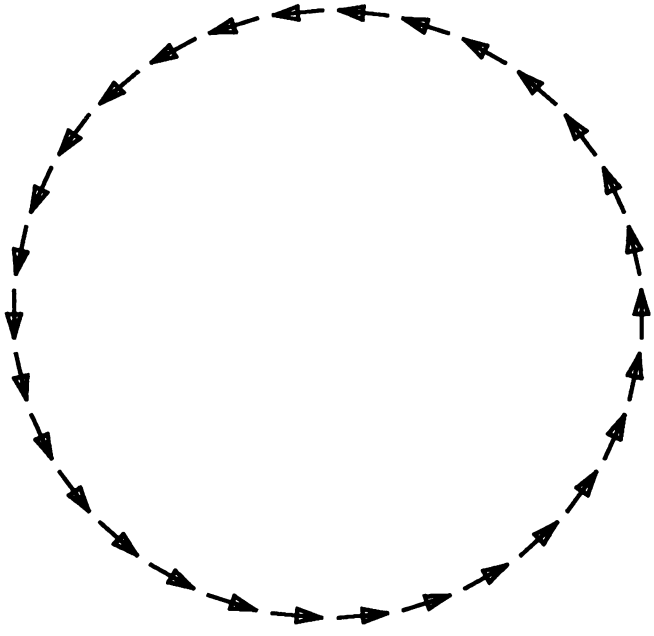


Fig. 2.5. Output of program ARROWS30

2.3 Matrix Notation

The two equations (2.2) can be written as the matrix equation

$$[x' \quad y'] = [x \quad y]\begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix} \tag{2.4}$$

or, with column vectors, as

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} \tag{2.5}$$

In books on computer graphics the row vector notation (2.4) is more frequently used than the column vector notation (2.5). We too will use notation (2.4). In this notation the ith row of the square matrix is always the image of the ith unit vector. It is possible to express Eqs. (2.3) as a matrix equation:

$$[x' \quad y'] = [x_0 \quad y_0] + [x-x_0 \quad y-y_0]\begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix} \tag{2.6}$$

However, the right-hand side of this equation is not a matrix product. In more complex situations, where rotations are combined with other transformations, it will be more convenient to have a single matrix product for each elementary transformation. At first sight this seems impossible in cases where translations are involved. However, it can be done if we do not insist on using only 2×2 transformation matrices, as we will see. We begin with a simple translation. Point $P(x, y)$ is translated to point $P'(x', y')$, where

$$\begin{aligned} x' &= x + a \\ y' &= y + b \end{aligned} \tag{2.7}$$

This could be written as

$$[x' \quad y'] = [x \quad y \quad 1]\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ a & b \end{bmatrix}$$

but in view of what will follow we prefer using a 3×3 transformation matrix, as occurs in the following matrix equation, which is equivalent to the above one and to Eqs. (2.7):

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix} \tag{2.8}$$

A technical term associated with this notation is *homogeneous coordinates*. Note that two-dimensional vectors are written with three real numbers. (In general, the homogeneous-coordinate notation $[x \quad y \quad w]$, where $w \neq 0$, denotes the vector that normally is written as $[x/w \quad y/w]$.)

Writing every transformation as a multiplication of matrices will enable us to combine several transformations into one. To show such a concatenation of transformations, we will combine a rotation with two translations. A rotation about O through an angle φ was described by Eq. (2.4). We replace this equation with

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

To describe a rotation about (x_0, y_0) through an angle φ, we will now replace Eq. (2.6) with the equation

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\, R \tag{2.10}$$

which does not use matrix addition. To find the 3×3 matrix R, occurring in this equation, we regard the transformation in question as a succession of the following three steps, with (u_1, v_1) and (u_2, v_2) as intermediate points:

(1) A translation to move (x_0, y_0) to O:

$$[u_1 \quad v_1 \quad 1] = [x \quad y \quad 1]\, T'$$

where

$$T' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_0 & -y_0 & 1 \end{bmatrix}$$

(2) A rotation through angle φ about O:

$$[u_2 \quad v_2 \quad 1] = [u_1 \quad v_1 \quad 1]\, R_0$$

where

$$R_0 = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.11}$$

(3) A translation from O to (x_0, y_0):

$$[x' \quad y' \quad 1] = [u_2 \quad v_2 \quad 1]\, T$$

where

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_0 & y_0 & 1 \end{bmatrix}$$

The combination of these three steps is based on the fact that matrix multiplication is associative, that is,

$$(AB)C = A(BC)$$

for any three matrices A, B and C whose dimensions are such that these multiplications are possible. For either side of this equation we simply write ABC. We then find

$$\begin{aligned} [x' \quad y' \quad 1] &= [u_2 \quad v_2 \quad 1]\, T \\ &= \{[u_1 \quad v_1 \quad 1]\, R_0\}\, T \\ &= [u_1 \quad v_1 \quad 1]\, R_0\, T \\ &= \{[x \quad y \quad 1]\, T'\}\, R_0\, T \\ &= [x \quad y \quad 1]\, T'\, R_0\, T \\ &= [x \quad y \quad 1]\, R \end{aligned}$$

where

$$R = T' R_0\, T$$

This is the desired matrix; performing two matrix multiplications gives:

$$R = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ c_1 & c_2 & 1 \end{bmatrix} \tag{2.12}$$

where c_1 and c_2 are the following constants:

$$c_1 = x_0 - x_0 \cos\varphi + y_0 \sin\varphi$$
$$c_2 = y_0 - x_0 \sin\varphi - y_0 \cos\varphi$$

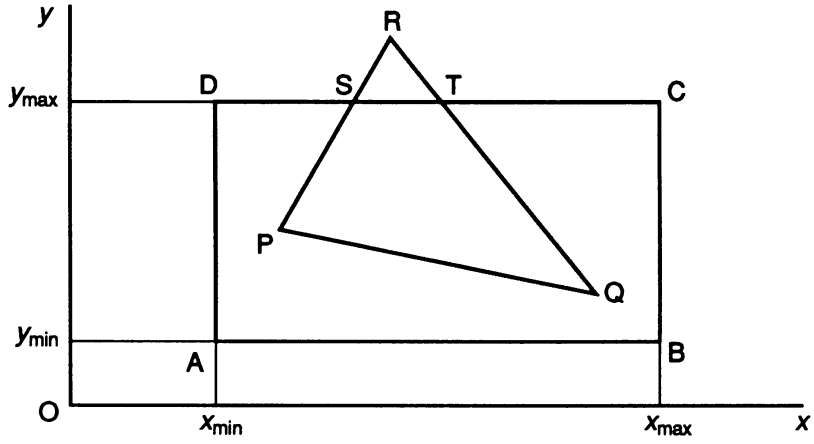


Fig. 2.6. Triangle to be clipped

2.4 Line Clipping

In this section we will discuss how to draw line segments only as far as they lie within a given rectangle with horizontal and vertical sides. We call such a rectangle a *window*, or sometimes, as we will see in Section 2.5, a *viewport*. Figure 2.6 shows window ABCD, together with triangle PQR, of which the line segments SR and TR must not be drawn because they lie outside the window. We want the clipping process to be done automatically. Commands to draw triangle PQR in Fig. 2.6 are to be interpreted as commands to draw only the line segments SP, PQ and QT. Since rectangle ABCD is drawn beforehand, we may say that the complete polygon PQTS is drawn instead of triangle PQR.

Since only the three points P, Q and R are given, the coordinate pair (x_S, y_S) has to be derived from (x_P, y_P) and (x_R, y_R). In Fig. 2.6 we see that the slope of PR can be expressed in two ways, which gives the following equation:

$$\frac{y_S - y_P}{x_S - x_P} = \frac{y_R - y_P}{x_R - x_P}$$

We combine this with

$$y_S = y_{max}$$

to find

$$x_S = x_P + \frac{(x_R - x_P)(y_{max} - y_P)}{(y_R - y_P)}$$

We can easily calculate the coordinates of S if, as in Fig. 2.6, endpoint P is known to be inside the window and endpoint $R(x_R, y_R)$ satisfies

$x_{min} < x_R < x_{max}$
$y_{max} < y_R$

However, there are many more cases to consider. The logical decisions needed to find out which actions are required make line clipping an interesting topic from an algorithmic point of view. For example, we have a completely different situation in Fig. 2.7, where it is clearly not sufficient to clip line segment PQ against line CD. Cohen and Sutherland developed an algorithm for line clipping, which is presented in Pascal by Newman and Sproull (1979) and by some other authors. We will now express this algorithm in the C++ language.

With any point P(x, y) we associate a four-bit code

$b_3\ b_2\ b_1\ b_0$

where b_i is either 0 or 1 (i = 0, 1, 2, 3). This code contains useful information about the position of P relative to window ABCD. In C++ the (truth) values of expressions such as **x < xmin** are 1 for 'true' and 0 for 'false'. Using this, we can write

```
b3 = (x < xmin);    // P to the left of AD
b2 = (x > xmax);    // P to the right of BC
b1 = (y < ymin);    // P below AB
b0 = (y > ymax);    // P above CD
```

Only nine out of the sixteen possible bit-configurations actually occur, as Fig. 2.8 shows. In C++ such a code value can be computed by using the operators << for left shift and | for bitwise 'or'. Instead of using four separate **int** variables, as just suggested, we can compute the four-bit code in a single return-statement, as the function **code** in the implementation module CLIP.CPP shows.

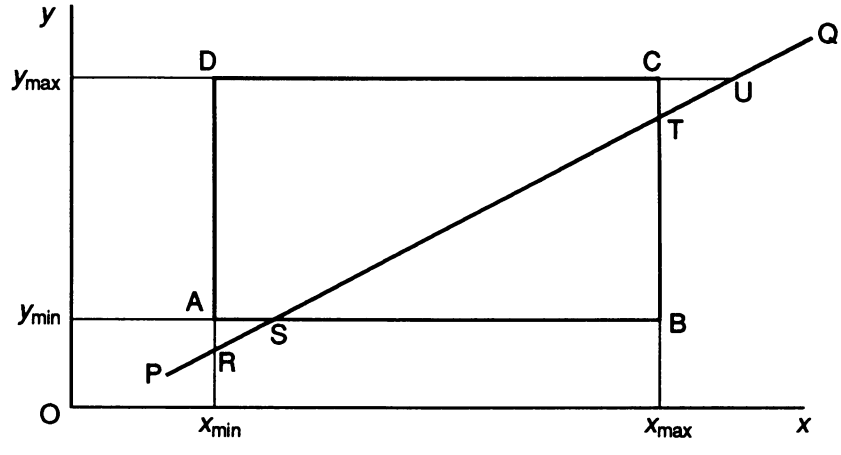


Fig. 2.7. Clipping in steps

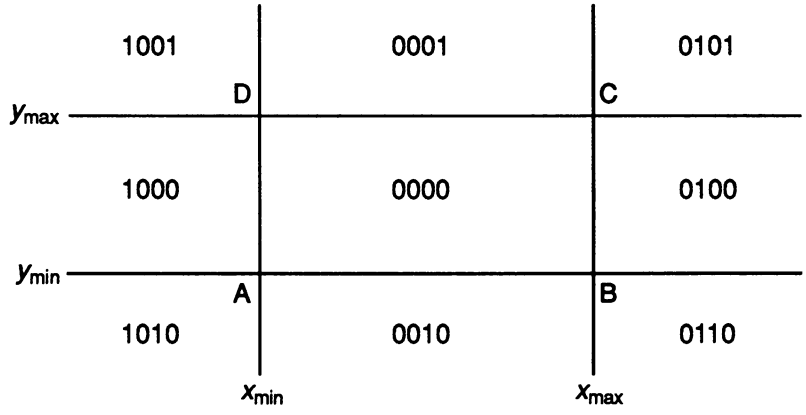


Fig. 2.8. Code values

```
// CLIP.CPP: Implementation module for Cohen-Sutherland line
//           clipping
#include "grsys.h"
#include "clip.h"
static float xmin, xmax, ymin, ymax;

void setclipboundaries(float x1, float x2, float y1, float y2)
{  xmin = x1; xmax = x2;
   ymin = y1; ymax = y2;
}

static int code(float x, float y)
{  return ((x<xmin)<<3) | ((x>xmax)<<2) |
          ((y<ymin)<<1) | (y>ymax);
}

void clipdraw(float xP, float yP, float xQ, float yQ)
{  int cP=code(xP, yP), cQ=code(xQ, yQ);
   float dx, dy;
   while (cP | cQ)
   {  if (cP & cQ) return;
      dx = xQ - xP; dy = yQ - yP;
      if (cP)
      {  if (cP & 8) yP += (xmin-xP)*dy/dx, xP=xmin; else
         if (cP & 4) yP += (xmax-xP)*dy/dx, xP=xmax; else
         if (cP & 2) xP += (ymin-yP)*dx/dy, yP=ymin; else
         if (cP & 1) xP += (ymax-yP)*dx/dy, yP=ymax;
         cP = code(xP, yP);
      }  else
      {  if (cQ & 8) yQ += (xmin-xQ)*dy/dx, xQ=xmin; else
```

```
            if (cQ & 4) yQ += (xmax-xQ)*dy/dx, xQ=xmax; else
            if (cQ & 2) xQ += (ymin-yQ)*dx/dy, yQ=ymin; else
            if (cQ & 1) xQ += (ymax-yQ)*dx/dy, yQ=ymax;
            cQ=code(xQ, yQ);
        }
    }
    move(xP, yP); draw(xQ, yQ);
}
```

As long as at least one of the codes for P and Q contains a 1-bit, either P or Q is moved from outside the window to one of its edges or to an extension of such an edge. In the latter case the point is still outside the window, so another move will be necessary. In Fig. 2.7, for example, a move from P to R must be followed by one from R to S. Then at the other end of the line segment clipping may also be necessary, as Fig. 2.7 shows. Thus clipping is a repetitive process; in each step the distance between P and Q decreases. The process terminates as soon as both points are no longer outside the window. Line segment PQ obtained in this way is then drawn. There is, however, another important case in which the loop is to terminate, namely if both P and Q are outside and on the same side of the window. This case may not apply initially but may arise during the clipping process. If the endpoints of a line segment are outside a window, the line segments may or may not intersect it, as Figs. 2.7 and 2.9 show.

In Fig. 2.9, initially P and Q are not both below the window. After two steps in the clipping process, R and S are the new positions of P and Q, respectively. Since both points are now below the window, it can be decided that nothing has to be drawn at all. Such decisions are made on the basis of the values of **code(xP, yP)** and **code(xQ, yQ)**. The points P and Q are on the same side of the window if and only if their codes have a 1 in the same position. For the three points P, R and S, the third bit (b_1) from the left in their codes is 1, whereas this bit is 0 for point Q . This is why Q has to be moved to S. Moving P to R is not necessary but this move is performed because it does no harm and keeps the algorithm simple.

Like the bitwise *or*-operator |, there is the bitwise *and*-operator, written as &. Remember, the bit operators & and | give bit strings as results, while their logical counterparts && and || yield only 0 or 1, which are the usual representations for 'false' and 'true', respectively. However, not only 1 but any nonzero value can act as 'true' if it is used in a logical context. Therefore the loop construction

```
while (c1 | c2) ...
```

will execute the actions indicated by ... as long as evaluation of **c1 | c2** gives a bit string containing a 1-bit, that is, as long as there is a 1-bit anywhere in **c1** or **c2**. On the other hand, the statement

```
if (c1 & c2) return;
```

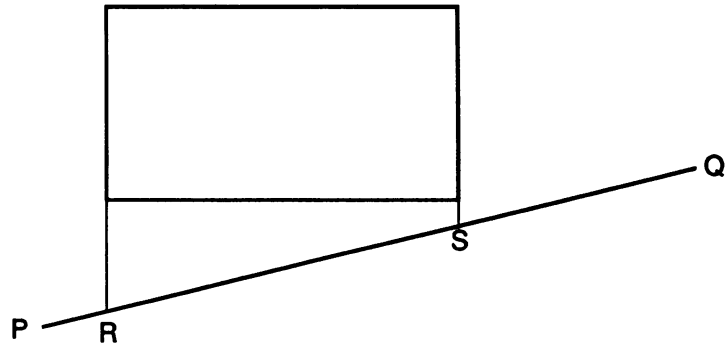


Fig. 2.9. Line outside window

causes a direct return from the function, if and only if the bit string value of **c1 & c2** contains a 1-bit, that is if and only if both **c1** and **c2** have a 1-bit in the same position.

When programming applications, we can use clipping without bothering about implementation details as we have just been discussing. We can use a very simple header file, shown below:

```
// CLIP.H: Header file for line clipping
void setclipboundaries(float x1, float x2, float y1, float y2);
void clipdraw(float xP, float yP, float xQ, float yQ);
```

Note that this header file was also used in an include-line in module CLIP.CPP. We could have omitted it there, but by including it any discrepancies between the function declarations and their definitions would be reported more clearly.

Function **setclipboundaries** is called only once, to set the static variables *xmin*, *xmax*, *ymin*, *ymax*, which are used later in function **clipdraw** which is called repeatedly. This is demonstrated by the following application module, which clips concentric pentagons:

```
/* CLIPDEMO: Demonstration of the Cohen & Sutherland
             line-clipping algorithm.
             To be linked together with modules CLIP and GRSYS
*/
#include <math.h>
#include "clip.h"
#include "grsys.h"

int main()
{  int i, j;
   float r, pi, alpha, phi0, phi, x1, y1, x2, y2, d,
         xmin, xmax, ymin, ymax;
   pi=4.0*atan(1.0); alpha=72.0*pi/180.0; phi0=0.0;
```

```
    initgr();
    d = 0.1 * r_max;
    xmin = x_min + d; xmax = x_max - d;
    ymin = y_min + d; ymax = y_max - d;
    // The window is now drawn:
    move(xmin, ymin); draw(xmax, ymin); draw(xmax, ymax);
    draw(xmin, ymax); draw(xmin, ymin);
    setclipboundaries(xmin, xmax, ymin, ymax);
    // As far as permitted by the boundaries of the window,
    // 20 concentric regular pentagons are drawn:
    for (j=1; j<=20; j++)
    {  r = j * d;
       x2 = x_center + r * cos(phi0);
       y2 = y_center + r * sin(phi0);
       for (i=1; i<=5; i++)
       {  phi = phi0 + i * alpha;
          x1 = x2; y1 = y2;
          x2 = x_center + r * cos(phi);
          y2 = y_center + r * sin(phi);
          clipdraw(x1, y1, x2, y2);
       }
    }
    endgr();
    return 0;
}
```

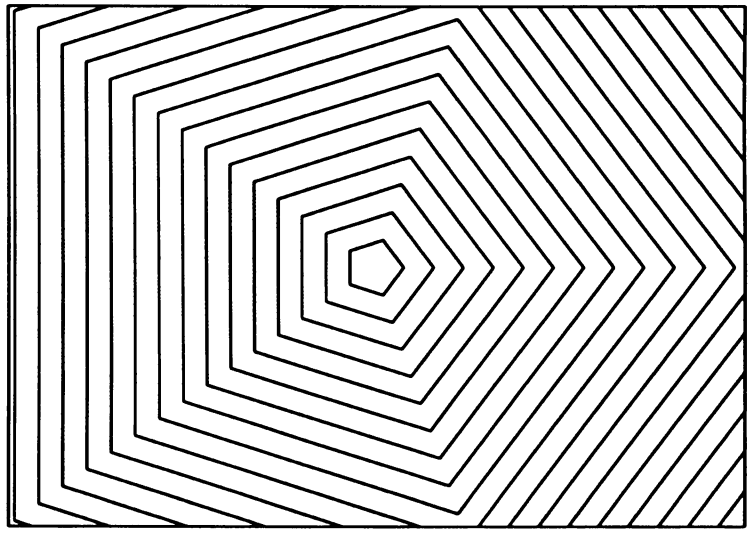


Fig. 2.10. Output of program CLIPDEMO

Figure 2.10 shows the output of this program. As you can see, there are only nine complete pentagons in the window; another nine are drawn incompletely because they lie partly outside the window. Since there are twenty calls to function clipdraw, two pentagons are not drawn at all because they lie completely outside the window.

2.5 Windows and Viewports

In many practical situations we have to draw objects whose dimensions are given in units completely incompatible with our screen coordinate system. A building may be a hundred times larger than its image. A molecule, on the other hand, is much smaller in reality than in a picture. Finally there are applications where the object to be drawn is not a concrete object but a graphical representation of relations between quantities, as, for example in Fig. 2.11, which shows the profits of a certain company at the beginning of the twentieth century.

Problem-oriented dimensions are expressed in so-called *world coordinates*. In Fig. 2.11 the numbers 1901, 1902, 1903, 1904, 50 000, 100 000, 150 000, 200 000 and 250 000 are world coordinates. We will now again use the concept of a *window*, which is a rectangle (with horizontal and vertical edges) surrounding the object to be displayed, as Fig. 2.11 shows. Note that, according to this terminology, a window is more closely related to the object than to the image to be produced. If, as usual, we introduce a horizontal x-axis and a vertical y-axis, the window in Fig. 2.11 is completely determined by

$$x_{min} = 1898 \qquad y_{min} = -50\,000$$
$$x_{max} = 1905 \qquad y_{max} = 325\,000$$

We see that the dimensions and the position of the window are expressed in world coordinates. This may surprise you, since the window was introduced to specify what we wish to see in the picture and, at first sight, a number of inches seems more natural to achieve this than, for example, a fictitious profit of –$ 50 000, as given here for y_{min}. However, expressing windows in world coordinates is customary and very convenient in practice.

To produce the desired picture, a rectangular region of the screen must be given as well. This region is called a *viewport*. It is specified like a window, that is, by the minimum and maximum values of the X- and Y-coordinates, but now they are screen coordinates. We will denote them by capital letters X and Y. A typical viewport specification is:

$$X_{min} = 1.5 \qquad Y_{min} = 1.0$$
$$X_{max} = 7.5 \qquad Y_{max} = 6.0$$

The window will now be mapped to the viewport. For example, with the window and the viewport just mentioned, the smallest world coordinate $x = 1898$ will be converted to the smallest screen coordinate $X = 1.5$.

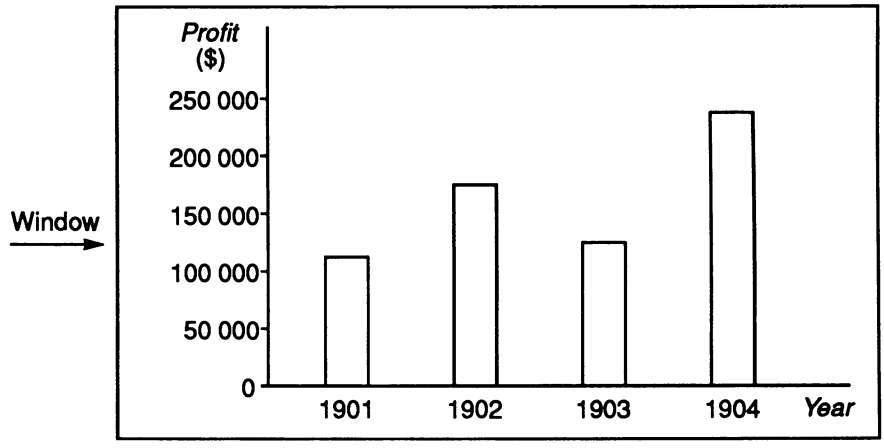


Fig. 2.11. Bar diagram in a window

First the scale factors f_x and f_y are computed:

$$f_x = \frac{X_{max} - X_{min}}{x_{max} - x_{min}}$$

$$f_y = \frac{Y_{max} - Y_{min}}{y_{max} - y_{min}}$$

Then the distance $X - X_{min}$ between an image point and the left viewport edge is found as f_x times the corresponding distance $x - x_{min}$ between the original point and the left window edge, and $Y - Y_{min}$ is found analogously, which leads to:

$$\begin{aligned} X &= X_{min} + f_x \cdot (x - x_{min}) \\ Y &= Y_{min} + f_y \cdot (y - y_{min}) \end{aligned} \tag{2.13}$$

We conclude this section with three remarks:

(1) The window may or may not surround the complete object. If it does not, the parts of the object which are outside the window must not be drawn but they are to be clipped off, as discussed in Section 2.4.

(2) The scale factors f_x and f_y may have different values. For a bar diagram this is just what we want, but it is not in cases where angles in the picture are to be the same as those in the object. We can then use the smaller of f_x and f_y for both scale factors. It is then recommended that Eq. (2.13) is replaced with formulae based on the centers of the window and the viewport, as we will see in Section 2.6.

(3) The dimensions and the position of the window are not always known beforehand. In Section 2.6 they will be calculated instead of specified by the user.

2.6 Uniform Scaling

To draw a picture of an object within the boundaries of a viewport, we can first clip it against a given window as in Section 2.4 and then map the window and its contents to the viewport, as in Section 2.5. For many applications this procedure is satisfactory, but for others it is not because we want to display the whole object rather than only part of it. In this section our approach will therefore be different:

(1) The object will be completely drawn, so no clipping will take place.
(2) The window will be computed rather than specified.
(3) In mapping from window to viewport the same scale factor will be applied to the horizontal and the vertical directions.

Point (1) requires that the object be finite; point (2) means that we do not know the window boundaries until all coordinates have been computed or read in. As these coordinates become available, we can keep track of their smallest and greatest values, x_{min}, x_{max}, y_{min} and y_{max}, which determine the window boundaries. Then, in a second phase, we can compute the screen (or viewport) coordinates since this time not only the viewport but also the window is known. We compute the scale factors f_x and f_y as in Section 2.5. However, this time we will use *uniform scaling*, which means that the same scale factor applies in the horizontal and vertical directions. Instead of using f_x and f_y themselves, we therefore use the smaller of them, which we will multiply by a reduction factor (≤ 1) to provide for blank margins along the four viewport edges. We therefore compute the factor

```
f = reductionfactor * (fx < fy ? fx : fy);
```

for uniform scaling, which we will use as follows:

$$X = X_{C'} + f \cdot (x - x_C)$$
$$Y = Y_{C'} + f \cdot (y - y_C)$$

As Fig. 2.12 illustrates, C and C′ denote the centers of the window and the viewport, respectively. In this example, a reduction factor $f = 0.5$ is used to map triangle PQR, lying in the window, to triangle P′Q′R′ in the viewport. The smallest rectangle (with a horizontal side), in which the object fits, is used here as the window, which means that x_{min} is the smallest of all the x-coordinates of the object, and x_{max} the largest. We can find y_{min} and y_{max} in the same way. We are also given the values X_{min}, X_{max}, Y_{min} and Y_{max}, which define the viewport. As you can see in Fig. 2.12, the object is drawn in the viewport in such a way that it touches two opposing edges. Because the above

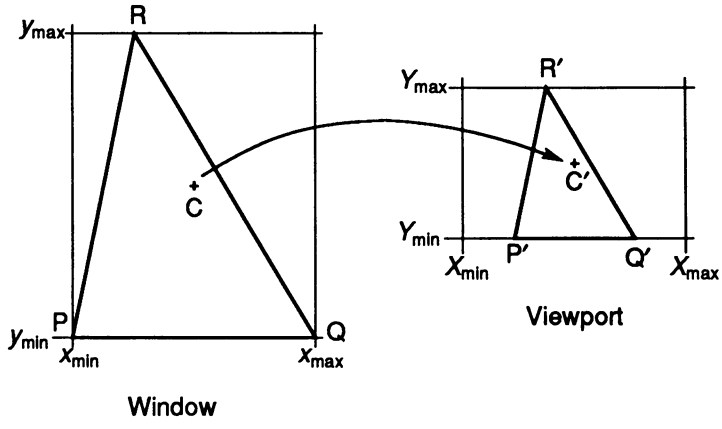


Fig. 2.12. Mapping from window to viewport, based on uniform scaling

formulae for uniform scaling map C to C′, the position of triangle P′Q′R′ in the viewport is *central*.

Determining the window boundaries can be done in two ways: either by computing (or reading) all coordinates twice in the same way, or by storing them somewhere in the first phase so that we can retrieve them in the second. Let us discuss each of these two principles by means of an example.

Reading (or computing) coordinates twice

Suppose we are given an input file with triples (*x*, *y*, *code*) where *code* is either 1, meaning 'pen down', or 0, meaning 'pen up'. Here is a very simple example of such a file, which corresponds to triangle PQR of Fig. 2.12.

```
11000  1000   0
15000  1000   1
12000  6000   1
11000  1000   1
```

We cannot use these *x* and *y* values directly, since they are world coordinates, and are to be converted to screen coordinates *X* and *Y*. A convenient way of doing this is based on reading this input file twice. The actual conversion will be done by the functions **x_viewport** and **y_viewport**, so that, after reading *x*, *y* and *code* for the second time, we can use the following code for the 'pen movement' that corresponds to this triple:

```
X = x_viewport(x); Y = y_viewport(y);
if (code) draw(X, Y); else move(X, Y);
```

Clearly **x_viewport** and **y_viewport** can do their work properly only if some data, obtained in or immediately after the first scan, is available. Let us introduce the functions **initwindow**, **updatewindowboundaries** and **viewportboundaries**, available in program module VIEWPORT. The following application module shows how to use them:

```
/* ADJUST: This program reads data from any text file
           containing lines of the form x y code
           (code = 1: pen down; code = 0: pen up), and
           displays lines, all fitting into a given viewport.
           The file name is to be supplied as a program argument.
           To be linked with GRSYS and VIEWPORT.
*/
#include <fstream.h>
#include <stdlib.h>
#include "grsys.h"
#include "viewport.h"

int main(int argc, char *argv[])
{  float x, y, X, Y; int code;
   ifstream inpfil(argv[1]);
   if (argc < 2 || !inpfil)
      errmess("No valid input file as program argument");
   initwindow();
   for ( ; ; )
   {  inpfil >> x >> y >> code;
      if (inpfil.eof() || inpfil.fail()) break;
      updatewindowboundaries(x, y);
   }
   inpfil.close();
   inpfil.open(argv[1]);
   initgr("adjust.hpg"); // Plot file adjust.hpg desired
   viewportboundaries(x_min, x_max, y_min, y_max, 0.9);
   move(x_min, y_min); draw(x_max, y_min);
   draw(x_max, y_max); draw(x_min, y_max);
   draw(x_min, y_min); // Show viewport
   for ( ; ; )
   {  inpfil >> x >> y >> code;
      if (inpfil.eof() || inpfil.fail()) break;
      X = x_viewport(x);
      Y = y_viewport(y);
      if (code) draw(X, Y); else move(X, Y);
   }
   endgr();
   return 0;
}
```

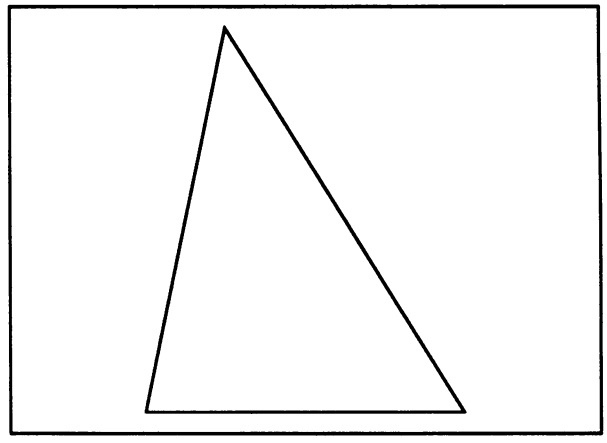


Fig. 2.13. Result of window-to-viewport mapping

The functions **initwindow**, **updatewindowboundaries**, **viewportboundaries** are to be called, in that order, before we call the functions **xviewport** and **yviewport**. These five functions are declared in the header file VIEWPORT.H, which we will discuss shortly. Program ADJUST also shows the use of the function **errmess**, which belongs to GRSYS and can be used to display an error message and stop program execution. (If there is a distinction between text and graphics modes, this function also switches back to text mode, in the same way as **endgr** normally does.)

Instead of using the whole screen, defined by the external variables *x_min*, *x_max*, *y_min*, *y_max* (available after calling **initgr**), program ADJUST uses only 90% of it; this is accomplished by means of the reduction factor 0.9, supplied as the final argument of **viewportboundaries**. If we link program ADJUST with module VIEWPORT, to be discussed below, the output is therefore as shown in Fig. 2.13. Note that the screen boundaries are also drawn so we can check that the triangle is really in the center of the screen.

The header file VIEWPORT.H, used in program ADJUST, has the following contents:

```
// VIEWPORT.H: Header file for viewport functions
void initwindow(void);
void updatewindowboundaries(float x, float y);
void viewportboundaries(float Xmin, float Xmax,
                        float Ymin, float Ymax,
                        float reductionfactor);
float x_viewport(float x);
float y_viewport(float y);
void append(float x, float y, char code);
void genplot(void);
```

Besides the five functions that we have already used, there are also two new ones, **append** and **genplot**, which we will discuss shortly. The implementation file VIEWPORT.CPP, listed below, shows the actual conversion from window to viewport coordinates, as discussed above and in Section 2.5:

```
/* VIEWPORT.CPP: Implementation of viewport functions
     Conversion from window coordinates (x, y) to
     viewport coordinates (X, Y)
*/
#include <stdlib.h>
#include "grsys.h"
#include "viewport.h"

const float BIG=1e30;
static float xmin, xmax, ymin, ymax,
             xC, yC, XC, YC, f;

static int iwcalled=0, wbcalled=0, vbcalled=0;

void initwindow(void)
{  xmin = ymin = BIG;
   xmax = ymax = -BIG;
   iwcalled = 1;
}

void updatewindowboundaries(float x, float y)
{  if (!iwcalled)
     errmess("Call 'initwindow' before 'updatewindowboundaries'");
   if (x < xmin) xmin = x;
   if (x > xmax) xmax = x;
   if (y < ymin) ymin = y;
   if (y > ymax) ymax = y;
   wbcalled = 1;
}

void viewportboundaries(float Xmin, float Xmax,
       float Ymin, float Ymax, float reductionfactor)
{  float fx, fy;
   if (!ingraphicsmode)
      errmess("Call 'initgr' before 'viewportboundaries'");
   if (!wbcalled) errmess("Call 'updatewindowboundaries' "
      "before 'viewportboundaries'");
   XC = 0.5 * (Xmin + Xmax);
   YC = 0.5 * (Ymin + Ymax);
   fx = (Xmax - Xmin)/(xmax - xmin + 1e-7);  // +1e-7 to prevent
   fy = (Ymax - Ymin)/(ymax - ymin + 1e-7);  // division by zero
```

```
   f = reductionfactor * (fx < fy ? fx : fy);
   xC = 0.5 * (xmin + xmax);
   yC = 0.5 * (ymin + ymax);
   vbcalled = 1;
}

float x_viewport(float x)
{  if (!vbcalled)
      errmess("Call 'viewportboundaries' before 'x_viewport'");
   return XC + f * (x - xC);
}

float y_viewport(float y)
{  return YC + f * (y - yC);
}

struct node
{  float x, y;
   char code;      // 0 = pen up; 1 = pen down
   node *next;
};

static node *start=NULL, *end=NULL;

void append(float x, float y, char code)
// Store point (x, y) and plotcode (0=up, 1=down) in queue.
{  node *old=end;
   end = new node;
   if (end == NULL) errmess("Not enough memory");
   end->x = x; end->y = y; end->code = code; end->next = NULL;
   updatewindowboundaries(x, y);
   if (start == NULL) start = end; else old->next = end;
}

void genplot(void)
{  float X, Y;
   node *ptr;
   if (!ingraphicsmode)
      errmess("Call 'initgr' before 'genplot'");
   viewportboundaries(x_min, x_max, y_min, y_max, 0.9);
   for (ptr=start; ptr!=NULL; ptr=ptr->next)
   {  X = x_viewport(ptr->x);
      Y = y_viewport(ptr->y);
      if (ptr->code) draw(X, Y); else move(X, Y);
   }
}
```

Storing coordinates in a queue

If the window coordinates are computed instead of read from an input file, we can determine the window boundaries by repeatedly calling **updatewindowboundaries** and store them somewhere so that they need not be computed again when we need them later. One possibility is to store them in a file on disk; another is to use a *linked list* in main memory for this purpose. The latter solution obviously will take an amount of memory space proportional to the number of points. If this is no problem, this method is attractive because it is faster than writing to and reading from disk. We will therefore demonstrate how this can be done. This linked list must be based on the principle 'First In First Out', that is, it must be a *queue*.

As an example, we will generate a curve of unpredictable shape and size, so that we do not know the window boundaries in advance. We will nevertheless draw this entire curve, in such a way that, either horizontally or vertically, it uses exactly 90% of the available space on the screen, with equal left and right margins and also with equal margins at the top and at the bottom. As before, there will be two scans. In the first, we compute coordinates x and y for the endpoint P of each elementary line segment and store them, together with a plot code, in a data structure, writing

```
append(x, y, code);
```

where *code* is either 0 or 1, meaning 'pen up' or 'pen down', respectively. This function **append** appends a new node containing x, y, and *code*, to the queue; it also updates the variables x_{min}, x_{max}, y_{min} and y_{max} by calling **updatewindowboundaries**. Although there are no explicit calls to latter function in our own program, we must prepare for such calls by using **initwindow** before the first call to **append**. Then after all coordinates have been stored by **append**, we simply use the call

```
genplot();
```

to draw the object in the viewport given by *x_min*, *x_max*, *y_min* and *y_max*. Recall that these variables are declared in the header file GRSYS.H and they have their correct values after a call to **initgr**. We have already seen the declarations and the definitions of **append** and **genplot** in the files VIEWPORT.H and VIEWPORT.CPP.

Our curve-generating application is based on random numbers. We start at the origin O and move one unit of distance at a time. There is always a current angle φ, which indicates the direction in which we are moving, and a current turning angle α. In each step, φ is increased by α. For example, if α has some positive value (less than 180°) then the current angle φ is increased by this positive value, which means that we turn to the left. Initially, φ and α are zero. In each step, another angle, θ, is chosen at random between –45° and +45°, that is, its number of degrees is one of the integers –45, –44,..., +45. Angle α is then increased by θ and subsequently divided by 2. The latter implies that we reduce the probability of α having the same sign for a long period of time, which would be undesirable because we do not want the curve to turn in one sense (either clockwise or counter-clockwise) in a great many successive steps. The following program generates a curve in this way:

```
/* CURV_GEN: Generation of a random curve.
             To be linked with GRSYS and VIEWPORT
*/
#include <math.h>
#include <iostream.h>
#include <stdlib.h>

#include "grsys.h"
#include "viewport.h"

int main()
{  int n;
   const float PIdiv180 = 3.1415926/180;
   int alpha=0, theta; // Angles in degrees
   float phi=0,        // Angle in radians
         x=0, y=0;
   cout << "How many line segments (e.g. 500): ";
   cin >> n;
   initwindow();
   append(x, y, 0);
   srand((unsigned int)time(NULL));
   while (n-- > 0)
   {  theta = rand() % 91 - 45;
      alpha = (alpha + theta)/2;
      phi += alpha * PIdiv180;
      x += cos(phi);
      y += sin(phi);
      append(x, y, 1);
   }
   initgr("curve.hpg"); // Plot file curve.hpg desired
   move(x_min, y_min);
   draw(x_max, y_min);
   draw(x_max, y_max);
   draw(x_min, y_max);
   draw(x_min, y_min); // Show viewport
   genplot();          // Plot curve, fitting into viewport
   endgr();
   return 0;
}
```

Note the calls to the functions **time** and **srand** just before the while-statement. The 'seed' for random-number generation, given as the argument of **srand**, depends on the actual clock time. In this way we will generate different curves if the program is executed more than once. Each time, function **rand** generates a large non-negative integer. This is converted to one of the integers 0, 1,..., 90 by taking the remainder after dividing it by 91. Thus we have

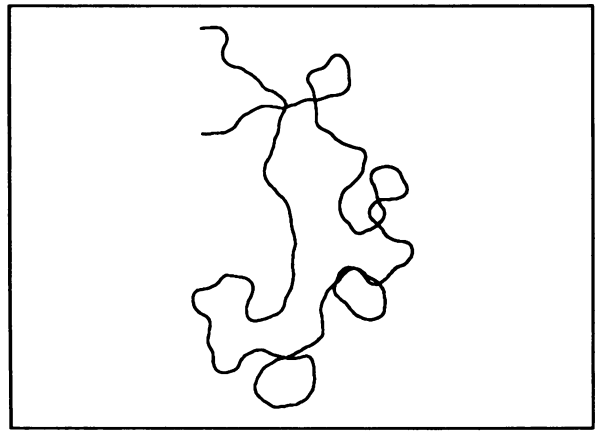


Fig. 2.14. A random curve, fitting into a given viewport

```
  0  ≤ rand( ) % 91      ≤  90
-45  ≤ rand( ) % 9 - 45  ≤ +45
```

An example of a random curve consisting of 500 line segments is shown in Fig. 2.14. As in Fig. 2.13, the large rectangle shows the screen boundaries.

A note on modular programming

The programs ADJUST and CURV_GEN are good examples of *modular programming*. This way of programming has the advantage that we can easily use modules for more than one application. The application modules ADJUST and CURV_GEN are rather short because they both use not only the general graphics module GRSYS, but also the more specific module VIEWPORT. Besides *reusability*, there is another advantage in using separate modules, namely that of *data hiding* by using static global variables. Using global variables is often more convenient and more efficient than using functions with long parameter lists. However, we can easily make mistakes if we use global variables in a large program that is written as one module. This is why global variables are often considered 'unsafe', especially by those who are used to writing large Pascal programs consisting of only one file. In C(++), however, we can use static global variables in small program modules, which gives us the best of both worlds: the convenience of using global variables is combined with safety. The keyword **static**, used when defining variables and functions makes it impossible for other modules to gain access to these variables and functions, so their scope is restricted to the file in which they occur. For example, there would be no problem at all if the identifier **start** were also used as a global name in module CURV_GEN.

Incidentally, many other languages, such as Fortran and assembly language (but not standard Pascal), offer similar facilities for modular programming.

2.7 Curve Fitting

In computer-aided design (CAD) and computer-aided manufacturing (CAM) it is often required to construct a smooth curve or a smooth surface through some given points. Here we will deal with two dimensions only, so we will restrict ourselves to curves in the xy-plane. Curve fitting will be a sound base for surface fitting at a later stage.[1]

Out of several available methods, we choose B-spline curve fitting as the one to be discussed and demonstrated here. A sequence of points is given, and between two successive points of this sequence we construct a cubic curve, based on the position of four *control points*, namely, the two points just mentioned and their two neighboring points. Consequently, moving a control point affects only a small part of the curve. This idea of *local control* is an interesting aspect of this method.

B-spline curves are very smooth; the price to be paid for this is that they do not exactly pass through the given points. The smoothness of a curve is mathematically expressed in terms of the continuity of its parametric representations $x(t)$ and $y(t)$ and their derivatives. Even the second derivatives $x''(t)$ and $y''(t)$ of B-splice curves are continuous at the points where two successive curve segments meet. In Fig. 2.15 we can see how curves look like if their zeroth, first or second derivatives are not continuous at some point. Although the curve in Fig. 2.15(c), consisting of a straight line and part of a circle, is generally considered smooth, it does not obey the strict rules imposed by the B-spline method.

After this informal discussion, we want to see this method in action. We will use a parametric representation of curves. Any point of a curve segment between two successive given points P and Q will have coordinates $x(t)$ and $y(t)$ where t increases from 0 to 1 if the curve segment is followed from P to Q. We can think of t as time. If we are given the points

$P_0(x_0, y_0)$
$P_1(x_1, y_1)$
$\vdots$
$P_n(x_n, y_n)$

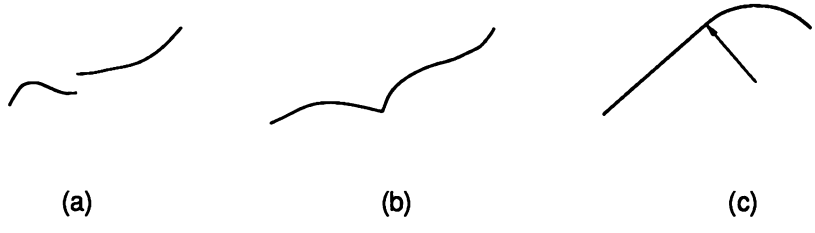


Fig. 2.15. (a) *0th,* (b) *1st, or* (c) *2nd derivative not continuous*

[1] There is a complete program for B-spline surface fitting in *Interactive 3D Computer Graphics* by the same author.

then the B-spline curve segment between two successive points P_i and P_{i+1} is obtained by computing $x(t)$ and $y(t)$, where t grows from 0 to 1:

$$\begin{aligned} x(t) &= \{(a_3t + a_2)t + a_1\}t + a_0 \\ y(t) &= \{(b_3t + b_2)t + b_1\}t + b_0 \end{aligned} \tag{2.14}$$

These equations contain the following coefficients:

$$\begin{aligned} a_3 &= (-x_{i-1} + 3x_i - 3x_{i+1} + x_{i+2})/6 & b_3 &= (-y_{i-1} + 3y_i - 3y_{i+1} + y_{i+2})/6 \\ a_2 &= (x_{i-1} - 2x_i + x_{i+1})/2 & b_2 &= (y_{i-1} - 2y_i + y_{i+1})/2 \\ a_1 &= (-x_{i-1} + x_{i+1})/2 & b_1 &= (-y_{i-1} + y_{i+1})/2 \\ a_0 &= (x_{i-1} + 4x_i + x_{i+1})/6 & b_0 &= (y_{i-1} + 4y_i + y_{i+1})/6 \end{aligned} \tag{2.15}$$

The formulae above are suitable for efficient computation. The value of $x(t)$ is given according to Horner's rule, rather than in the usual notation for polynomials. The eight coefficients a_j and b_j are computed only once for each of the curve segments. This is most important, since we want to compute $x(t)$ and $y(t)$ a great many times on a single curve segment.

Derivation of B-spline coefficients

Before we put all this into practice, let us see how the formulae (2.14) and (2.15) can be derived. First, note that we can write Eqs. (2.14) as the matrix product

$$\begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_3 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}$$

and Eqs. (2.15) as

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_3 \end{bmatrix} = \frac{1}{6} \begin{bmatrix} x_{i-1} & x_i & x_{i+1} & x_{i+2} \\ y_{i-1} & y_i & y_{i+1} & y_{i+2} \end{bmatrix} \begin{bmatrix} 1 & -3 & 3 & -1 \\ 4 & 0 & -6 & 3 \\ 1 & 3 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which can be combined into

$$\begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = \frac{1}{6} \begin{bmatrix} x_{i-1} & x_i & x_{i+1} & x_{i+2} \\ y_{i-1} & y_i & y_{i+1} & y_{i+2} \end{bmatrix} \begin{bmatrix} 1 & -3 & 3 & -1 \\ 4 & 0 & -6 & 3 \\ 1 & 3 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix} \tag{2.16}$$

Actually, the 4 × 4 matrix occurring in this equation is still to be derived. Let us write this matrix, multiplied by the above factor 1/6, as

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

and use the abbreviations

$$\mathbf{p}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} \qquad \mathbf{p}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Then Eq. (2.16) is written as

$$\mathbf{p}(t) = [\mathbf{p}_{i-1} \;\; \mathbf{p}_i \;\; \mathbf{p}_{i+1} \;\; \mathbf{p}_{i+2}]\, A\, [1 \;\; t \;\; t^2 \;\; t^3]^T \tag{2.17}$$

Note that this is a vector-valued function with a real argument t ranging from 0 to 1; it is our task to choose the sixteen elements of matrix A such that this function satisfies some conditions. There is such a function for each part of the curve between any two successive points (except for the first and the last) of the given sequence of points. For example, if we are given the five points P, Q, R, S, T, then there are two such functions: one for the part of the curve between Q and R and one for that between R and S. Let us denote these functions by $\mathbf{p}_{QR}(t)$ and $\mathbf{p}_{RS}(t)$, respectively. We pay special attention to the situation near point R. Continuity of the whole curve requires the endpoint of the first part of the curve to coincide with the starting point of the second; in other words, we have

$$\mathbf{p}_{QR}(1) = \mathbf{p}_{RS}(0) \tag{2.18}$$

If $\mathbf{p}_{i-1}$, $\mathbf{p}_i$, $\mathbf{p}_{i+1}$, $\mathbf{p}_{i+2}$ are the four given points related to $\mathbf{p}_{QR}(t)$, then $\mathbf{p}_i$, $\mathbf{p}_{i+1}$, $\mathbf{p}_{i+2}$, $\mathbf{p}_{i+3}$ are those related to $\mathbf{p}_{RS}(t)$. Using (2.17), we can therefore write (2.18) as the following equation, in which A is the above 4 × 4 matrix, and the left-hand and the right-hand sides of the equation are 2 × 1 matrices:

$$[\mathbf{p}_{i-1} \;\; \mathbf{p}_i \;\; \mathbf{p}_{i+1} \;\; \mathbf{p}_{i+2}]\, A\, [1 \;\; 1 \;\; 1 \;\; 1]^T = [\mathbf{p}_i \;\; \mathbf{p}_{i+1} \;\; \mathbf{p}_{i+2} \;\; \mathbf{p}_{i+3}]\, A\, [1 \;\; 0 \;\; 0 \;\; 0]^T$$

This can also be written as the vector equation

$$\begin{aligned} &(a_{11} + a_{12} + a_{13} + a_{14})\mathbf{p}_{i-1} \\ &+ (a_{21} + a_{22} + a_{23} + a_{24})\mathbf{p}_i \\ &+ (a_{31} + a_{32} + a_{33} + a_{34})\mathbf{p}_{i+1} \\ &+ (a_{41} + a_{42} + a_{43} + a_{44})\mathbf{p}_{i+2} \;=\; a_{11}\mathbf{p}_i + a_{21}\mathbf{p}_{i+1} + a_{31}\mathbf{p}_{i+2} + a_{41}\mathbf{p}_{i+3} \end{aligned}$$

The elements a_{ij} of matrix A are constants, which do not depend on the given points. Consequently, the coefficient of $\mathbf{p}_i$ on the left-hand side of the equation above must be equal to the coefficient of $\mathbf{p}_i$ in the right-hand side, and so on. We therefore have

$$\begin{aligned}
a_{11} + a_{12} + a_{13} + a_{14} &= 0 && \text{(coefficients of } \mathbf{p}_{i-1}) \\
a_{21} + a_{22} + a_{23} + a_{24} &= a_{11} && \text{(coefficients of } \mathbf{p}_i) \\
a_{31} + a_{32} + a_{33} + a_{34} &= a_{21} && \text{(coefficients of } \mathbf{p}_{i+1}) \\
a_{41} + a_{42} + a_{43} + a_{44} &= a_{31} && \text{(coefficients of } \mathbf{p}_{i+2}) \\
0 &= a_{41} && \text{(coefficients of } \mathbf{p}_{i+3})
\end{aligned}$$

Since matrix A has sixteen elements, we must have sixteen equations to compute them, and we now have only five. Remembering our discussion about continuity, illustrated by Fig. 2.15, we proceed as follows. In addition to applying the continuity restriction to the functions $\mathbf{p}(t)$ themselves, as we just have done, we now also apply them to their first and second derivatives. By differentiating $\mathbf{p}(t)$ twice, we find

$$\mathbf{p}'(t) = [\mathbf{p}_{i-1} \;\; \mathbf{p}_i \;\; \mathbf{p}_{i+1} \;\; \mathbf{p}_{i+2}]\, A\, [0 \;\; 1 \;\; 2t \;\; 3t^2]^{\mathrm{T}}$$

$$\mathbf{p}''(t) = [\mathbf{p}_{i-1} \;\; \mathbf{p}_i \;\; \mathbf{p}_{i+1} \;\; \mathbf{p}_{i+2}]\, A\, [0 \;\; 0 \;\; 2 \;\; 6t]^{\mathrm{T}}$$

In the same way as Eq. (2.18) proved to be very useful, the equations

$$\mathbf{p}'_{\mathrm{QR}}(1) = \mathbf{p}'_{\mathrm{RS}}(0) \qquad \text{and} \qquad \mathbf{p}''_{\mathrm{QR}}(1) = \mathbf{p}''_{\mathrm{RS}}(0)$$

lead to

$$\begin{aligned}
a_{12} + 2a_{13} + 3a_{14} &= 0 && \text{(coefficients of } \mathbf{p}_{i-1}) \\
a_{22} + 2a_{23} + 3a_{24} &= a_{12} && \text{(coefficients of } \mathbf{p}_i) \\
a_{32} + 2a_{33} + 3a_{34} &= a_{22} && \text{(coefficients of } \mathbf{p}_{i+1}) \\
a_{42} + 2a_{43} + 3a_{44} &= a_{32} && \text{(coefficients of } \mathbf{p}_{i+2}) \\
0 &= a_{42} && \text{(coefficients of } \mathbf{p}_{i+3})
\end{aligned}$$

and

$$\begin{aligned}
2a_{13} + 6a_{14} &= 0 && \text{(coefficients of } \mathbf{p}_{i-1}) \\
2a_{23} + 6a_{24} &= 2a_{13} && \text{(coefficients of } \mathbf{p}_i) \\
2a_{33} + 6a_{34} &= 2a_{23} && \text{(coefficients of } \mathbf{p}_{i+1}) \\
2a_{43} + 6a_{44} &= 2a_{33} && \text{(coefficients of } \mathbf{p}_{i+2}) \\
0 &= a_{43} && \text{(coefficients of } \mathbf{p}_{i+3})
\end{aligned}$$

respectively. We now have fifteen equations in sixteen unknowns a_{kj}, so we still need one. Note that all equations found so far are homogeneous: if a set of values a_{kj} satisfies them, then so does the set of values λa_{kj} for any real number λ. (In particular, there is the zero solution, corresponding to $\lambda = 0$.) Curiously enough, we have been dealing only with aspects of continuity, but we have not yet used any requirement that the curve approximates to the given points as well as is possible! Consequently, if we

used just any solution to the above system of fifteen equations, we might find some curve that, though perfectly satisfying our continuity requirements, lies very far away from the given points. The remaining equation must remedy this. We now substitute $t = 0$ in Eq. (2.17), which gives

$$\mathbf{p}(0) = a_{11}\mathbf{p}_{i-1} + a_{21}\mathbf{p}_i + a_{31}\mathbf{p}_{i+1} + a_{41}\mathbf{p}_{i+2}$$

If no other information were available, we might make $\mathbf{p}(0)$ coincide with $\mathbf{p}_i$ by setting $a_{21} = 1$ and $a_{11} = a_{31} = a_{41} = 0$. Another possibility would be to use 0.25 for all these four coefficients. In either case, we would have

$$a_{11} + a_{21} + a_{31} + a_{41} = 1$$

This non-homogeneous equation is just the one we need: it does not violate the fifteen other equations and it makes the curve a good approximation of the given points.

We now have sixteen linear equations in sixteen unknowns. Solving these gives the desired result

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \frac{1}{6}\begin{bmatrix} 1 & -3 & 3 & -1 \\ 4 & 0 & -6 & 3 \\ 1 & 3 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

So much for this rather technical derivation of the B-spline coefficients.

A programming example

Program CURVEFIT reads the numbers

$$\begin{array}{ll} n & \\ x_0 & y_0 \\ x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{array}$$

from the file CURVEFIT.TXT. In the output each of these $n+1$ points (x_i, y_i) is plotted in the form of a cross, after which the B-spline curve is drawn. The program uses the technique discussed in Section 2.6, so the curve fits nicely into the screen boundaries, regardless of the given coordinate values. Since these coordinates are stored in an array, there is no need to read the data twice from an input file (as was done in program ADJUST); nor do we need a queue (as in program CURV_GEN).

```
// CURVEFIT: Curve fitting using B splines.
//     To be linked with GRSYS and VIEWPORT.
#include <fstream.h>
#include <stdlib.h>
#include "grsys.h"
#include "viewport.h"

const int MAX=1000, N=30;
// At most MAX points and N intervals between any two
// neighbor points

int main()
{  float x[MAX], y[MAX], eps=0.04, X, Y, t, xA, xB, xC, xD,
        yA, yB, yC, yD, a0, a1, a2, a3, b0, b1, b2, b3;
   int n, i, j, first;
   ifstream inpfil("curvefit.txt");
   if (!inpfil)
      errmess("Cannot open file curvefit.txt for input");
   inpfil >> n;
   if (n < 3 || n+1 >= MAX)
      errmess("First number read incorrect");
   initwindow();
   for (i=0; i<=n; i++)
   {  inpfil >> x[i] >> y[i];
      if (inpfil.eof() || inpfil.fail())
         errmess("Error in input file");
      updatewindowboundaries(x[i], y[i]);
   }

   initgr();
   viewportboundaries(x_min, x_max, y_min, y_max, 0.9);
   // Mark the given points:
   for (i=0; i<=n; i++)
   {  X = x_viewport(x[i]);
      Y = y_viewport(y[i]);
      move(X-eps, Y-eps); draw(X+eps, Y+eps);
      move(X+eps, Y-eps); draw(X-eps, Y+eps);
   }
   first=1;
   for (i=1; i<n-1; i++)
   {  xA=x[i-1]; xB=x[i]; xC=x[i+1]; xD=x[i+2];
      yA=y[i-1]; yB=y[i]; yC=y[i+1]; yD=y[i+2];
      a3=(-xA+3*(xB-xC)+xD)/6.0; b3=(-yA+3*(yB-yC)+yD)/6.0;
      a2=(xA-2*xB+xC)/2.0;       b2=(yA-2*yB+yC)/2.0;
      a1=(xC-xA)/2.0;            b1=(yC-yA)/2.0;
      a0=(xA+4*xB+xC)/6.0;       b0=(yA+4*yB+yC)/6.0;
```

```
        for (j=0; j<=N; j++)
        {  t = (float)j/(float)N;
           X = x_viewport(((a3*t+a2)*t+a1)*t+a0);
           Y = y_viewport(((b3*t+b2)*t+b1)*t+b0);
           if (first) {first=0; move(X, Y);} else draw(X, Y);
        }
     }
     endgr(); return 0;
}
```

The input file CURVEFIT.TXT contains the following data, the output is as shown in Fig. 2.16.

```
20
0.25    0.4
0       0
0.25   -0.4
1      -0.6
1.75   -0.3
2      -0.1
2.5    -0.15
4.5    -0.25
5.75   -0.2
6      -0.15
6       0
6       0.15
5.75    0.2
4.5     0.25
2.5     0.15
2       0.1
1.75    0.3
1       0.6
0.25    0.4
0       0
0.25   -0.4
```

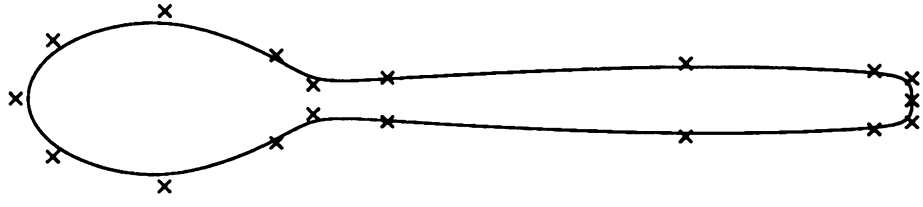


Fig. 2.16. Output of program CURVEFIT

Exercises

2.1 Write a program to draw a set of N ellipses whose parametric equations are

$$x = x_0 + (iR/N) \cos \varphi$$
$$y = y_0 + (N - i)(R/N) \sin \varphi$$

Choose suitable values x_0, y_0, R, N. Let i range from 1 to $N - 1$. For each value i, let φ successively have the values 0°, 6°, 12°,..., 360°.

2.2 Write a program to draw a set of 30 triangles. For the first triangle, the coordinates of its vertices are read in, and so are the coordinates of a point P. Except for the first, each triangle is to be obtained by rotating the previous one about P through an angle of 3°.

2.3 Implement a recursive line-clipping algorithm based on bisection. Suppose a window and a line segment PQ are given. If P and Q are both inside the window, PQ can be drawn. This is also the case if one of these points is inside the window and the other is on a window edge. There are also some cases where we can easily decide that nothing is to be drawn. Find these yourself. In all other cases, find point M in the middle of PQ, and apply the same procedure recursively to PM and MQ. Choose a tolerance with respect to the window edges, and investigate the influence of this tolerance on the number of bisections.

2.4 Develop and implement an algorithm for clipping a line against a triangle.

2.5 Use random numbers to generate a sequence of, say, 30 points, and apply curve fitting to draw a curve (approximately) through these points.

3

Geometric Tools

3.1 Vectors and Coordinate Systems

Before we proceed with specific graphics subjects, let us briefly discuss some mathematics, which we will often use in this book. Some of the subjects discussed in this chapter can also be found in mathematics textbooks.

Although we have already used vectors (in two-dimensional space), a definition of this notion will not be omitted here. This definition also applies to three-dimensional space, and it is 'geometric'. A *vector* is a directed line segment, characterized by its length and its direction only. Figure 3.1 shows two representations of the same vector $\mathbf{a} = \mathbf{PQ} = \mathbf{b} = \mathbf{RS}$. Thus a vector is not altered by a translation. In Fig. 3.2 the start point of **b** is the end point of **a**. Then the sum of **a** and **b** can be defined as the vector **c** drawn from the start point of **a** to the end point of **b**, and we write

$$\mathbf{c} = \mathbf{a} + \mathbf{b}$$

The length of a vector **a** is denoted by $|\mathbf{a}|$. A vector with zero length is the zero vector, written **0**. The notation $-\mathbf{a}$ is used for the vector that has length $|\mathbf{a}|$ and whose direction is opposite to that of **a**. For any vector **a** and real number c, the vector $c\mathbf{a}$ has length $|c||\mathbf{a}|$. If $\mathbf{a} = \mathbf{0}$ or $c = 0$, then $c\mathbf{a} = \mathbf{0}$; otherwise $c\mathbf{a}$ has the direction of **a** if $c > 0$ and the opposite direction if $c < 0$. For any vectors **u**, **v**, **w** and real numbers c, k, we have

$$\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$$
$$(\mathbf{u} + \mathbf{v}) + \mathbf{w} = \mathbf{u} + (\mathbf{v} + \mathbf{w})$$
$$\mathbf{u} + \mathbf{0} = \mathbf{u}$$
$$\mathbf{u} + (-\mathbf{u}) = \mathbf{0}$$

$c(\mathbf{u} + \mathbf{v}) = c\mathbf{u} + c\mathbf{v}$
$(c + k)\mathbf{u} = c\mathbf{u} + k\mathbf{u}$
$c(k\mathbf{u}) = (ck)\mathbf{u}$
$1\mathbf{u} = \mathbf{u}$
$0\mathbf{u} = \mathbf{0}$

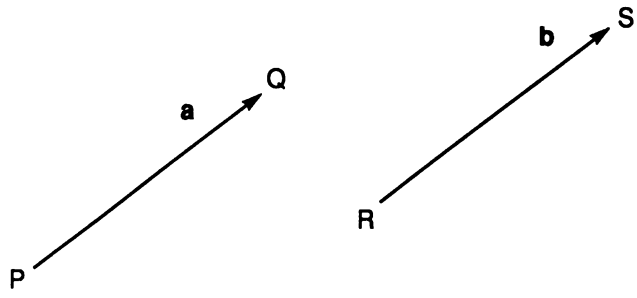


Fig. 3.1. Two equal vectors

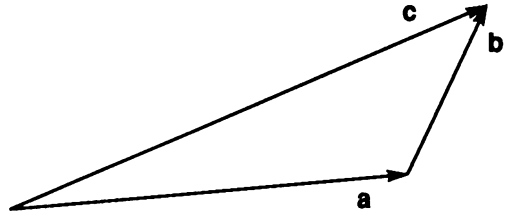


Fig. 3.2. Vector addition

Figure 3.3 shows three unit vectors **i**, **j** and **k**. They are mutually perpendicular and have length 1. Their directions are the positive directions of the coordinate axes. We say that **i**, **j** and **k** form a triple of orthogonal *unit vectors*. The coordinate system is right-handed, which means that if a rotation of **i** in the direction of **j** through 90° corresponds to turning a right-handed screw, then **k** has the direction in which the screw advances.

We often choose the origin O of the coordinate system as the initial point of all vectors. Any vector **v** can be written as a linear combination of the unit vectors **i**, **j**, and **k**:

$$\mathbf{v} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

The real numbers x, y and z are the coordinates of the end point P of vector $\mathbf{v} = \mathbf{OP}$. We often write this vector **v** as

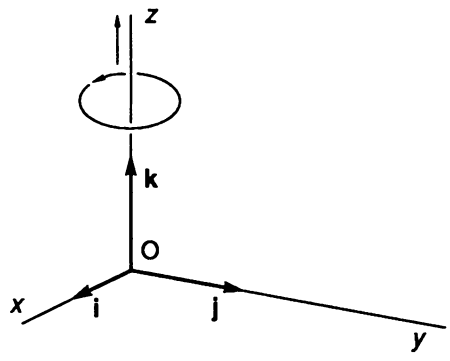


Fig. 3.3. Right-handed coordinate system

$\mathbf{v} = [x \quad y \quad z]$ or as $\mathbf{v} = (x, y, z)$

The numbers x, y, z are sometimes called the *elements* or *components* of vector **v**.

3.2 Inner Product

The *inner product*, *dot product*, or *scalar product* of two vectors **a** and **b** is written $\mathbf{a} \cdot \mathbf{b}$ and is defined as

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= |\mathbf{a}||\mathbf{b}|\cos\gamma && \text{if } \mathbf{a} \neq 0 \text{ and } \mathbf{b} \neq 0 \\ \mathbf{a} \cdot \mathbf{b} &= 0 && \text{if } \mathbf{a} = 0 \text{ or } \mathbf{b} = 0 \end{aligned} \tag{3.1}$$

where γ is the angle between **a** and **b**. Applying this to the unit vectors **i**, **j** and **k**, we find

$$\begin{aligned} &\mathbf{i} \cdot \mathbf{i} = \mathbf{j} \cdot \mathbf{j} = \mathbf{k} \cdot \mathbf{k} = 1 \\ &\mathbf{i} \cdot \mathbf{j} = \mathbf{j} \cdot \mathbf{i} = \mathbf{j} \cdot \mathbf{k} = \mathbf{k} \cdot \mathbf{j} = \mathbf{k} \cdot \mathbf{i} = \mathbf{i} \cdot \mathbf{k} = 0 \end{aligned} \tag{3.2}$$

Setting $\mathbf{b} = \mathbf{a}$ in Eqs. (3.1), we have $\mathbf{a} \cdot \mathbf{a} = |\mathbf{a}|^2$, so

$$|\mathbf{a}| = \sqrt{|\mathbf{a} \cdot \mathbf{a}|}$$

Some important properties of inner products are

$$\begin{aligned} &c(k\mathbf{u} \cdot \mathbf{v}) = ck(\mathbf{u} \cdot \mathbf{v}) \\ &(c\mathbf{u} + k\mathbf{v}) \cdot \mathbf{w} = c\mathbf{u} \cdot \mathbf{w} + k\mathbf{v} \cdot \mathbf{w} \\ &\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u} \\ &\mathbf{u} \cdot \mathbf{u} = 0 \text{ only if } \mathbf{u} = \mathbf{0} \end{aligned}$$

The inner product of two vectors $\mathbf{u} = [u_1 \ u_2 \ u_3]$ and $\mathbf{v} = [v_1 \ v_2 \ v_3]$ can be computed as

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + u_3 v_3$$

We can prove this by developing the right-hand side of the following equality as the sum of nine inner products and then applying Eqs. (3.2):

$$\mathbf{u} \cdot \mathbf{v} = (u_1 \mathbf{i} + u_2 \mathbf{j} + u_3 \mathbf{k}) \cdot (v_1 \mathbf{i} + v_2 \mathbf{j} + v_3 \mathbf{k})$$

3.3 Determinants and Orientation

Before proceeding with vector products, we will pay some attention to determinants. Suppose we want to solve the following system of two linear equations for x and y:

$$\begin{cases} a_1 x + b_1 y = c_1 \\ a_2 x + b_2 y = c_2 \end{cases} \tag{3.3}$$

We can then multiply the first equation by b_2, the second by $-b_1$, and add them up, finding

$$(a_1 b_2 - a_2 b_1) x = b_2 c_1 - b_1 c_2$$

Then we can multiply the first equation by $-a_2$, the second by a_1, and add again, finding

$$(a_1 b_2 - a_2 b_1) y = a_1 c_2 - a_2 c_1$$

If $a_1 b_2 - a_2 b_1$ is not zero, we can divide and find

$$x = \frac{b_2 c_1 - b_1 c_2}{a_1 b_2 - a_2 b_1} \qquad y = \frac{a_1 c_2 - a_2 c_1}{a_1 b_2 - a_2 b_1} \tag{3.4}$$

The denominator in these expressions is often written in the form

$$\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}$$

and then called a *determinant*. Thus

$$\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = a_1 b_2 - a_2 b_1$$

Using determinants, we can write the solution of Eqs. (3.3) as

$$x = \frac{D_1}{D} \qquad y = \frac{D_2}{D} \qquad (D \neq 0)$$

where

$$D = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} \qquad D_1 = \begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix} \qquad D_2 = \begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}$$

Note that D_i is obtained by replacing the ith column of D with the right-hand side of Eqs. (3.3) ($i = 1$ or 2). This method of solving a system of linear equations is called *Cramer's rule*. It is not restricted to systems of two equations (although it would be very expensive in terms of computer time to apply the method to large systems). Determinants with n rows and n columns are said to be of the nth order. We define determinants of third order by the equation

$$D = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}M_{11} - a_{12}M_{12} + a_{13}M_{13}$$

where each so-called *minor determinant* M_{ij} is the 2×2 determinant that we obtain by deleting the ith row and the jth column of D. Determinants of higher order are defined similarly. For example, determinants of fourth order are defined by

$$D = \begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{vmatrix} = a_{11}M_{11} - a_{12}M_{12} + a_{13}M_{13} - a_{14}M_{14}$$

where each 3×3 minor determinant M_{ij} is obtained by deleting the ith row and the jth column of D.

Determinants are very useful in linear algebra and analytical geometry. They have many interesting properties, some of which are listed below:

(1) The value of a determinant remains the same if its rows are written as columns in the same order. For example:

$$\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix}$$

(2) If any two rows (or columns) are interchanged, the value of the determinant is multiplied by -1. For example:

$$\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} = - \begin{vmatrix} a_1 & b_1 & c_1 \\ a_3 & b_3 & c_3 \\ a_2 & b_2 & c_2 \end{vmatrix}$$

(3) If any row or column is multiplied by a factor, the value of the determinant is multiplied by this factor. For example:

$$\begin{vmatrix} ca_1 & b_1 \\ ca_2 & b_2 \end{vmatrix} = c \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix}$$

(4) If a row is altered by adding any constant multiple of any other row to it, the value of the determinant remains unaltered. This operation may also be applied to columns. For example:

$$\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3+ka_1 & b_3+kb_1 & c_3+kc_1 \end{vmatrix} = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}$$

(5) If a row (or a column) is a linear combination of some other rows (or columns), the value of the determinant is zero. For example:

$$\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ 3a_1-2a_2 & 3b_1-2b_2 & 3c_1-2c_2 \end{vmatrix} = 0$$

There are many useful applications of determinants. In many cases, determinant equations expressing geometrical properties are elegant and easy to remember. For example, the equation of the line in R_2 through the two points $P_1(x_1, y_1)$, $P_2(x_2, y_2)$ can be written

$$\begin{vmatrix} x & y & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix} = 0 \tag{3.5}$$

This can be understood by observing, first, that Eq. (3.5) is a special notation for a linear equation in x and y, and consequently represents a straight line in R_2, and second, that the coordinates of both P_1 and P_2 satisfy this equation, for if we write them in the first row, we have two identical rows. Similarly, the plane in R_3 through the three points $P_1(x_1, y_1, z_1)$, $P_2(x_2, y_2, z_2)$, $P_3(x_3, y_3, z_3)$ has the following equation, which is much easier to remember than one written in the conventional way:

$$\begin{vmatrix} x & y & z & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} = 0$$

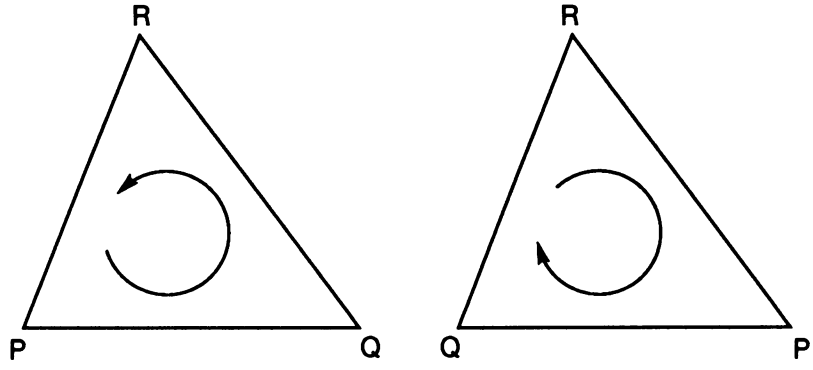


Fig. 3.4. Counter-clockwise and clockwise orientation of P, Q, R

The orientation of three points

We will now deal with a subject that will be very useful in Section 3.5 and in Chapters 6 and 7. Suppose that we are given an ordered triple (P, Q, R) of three points in the xy-plane and we want to know their orientation; in other words, we want to know whether we turn counter-clockwise or clockwise when visiting these points in the given order. Figure 3.4 shows the possibilities, which we refer to as *positive* and *negative* orientation, respectively. There is a third possibility, namely that the points P, Q, and R lie on a straight line. We will consider the orientation to be zero in this case. If we plot the points on paper, we see immediately which of these three cases applies, but we now want a means to find the orientation by computation, using only the coordinates x_P, y_P, x_Q, y_Q, x_R, y_R.

Since a translation of three points does not alter their orientation, we may as well decrease the three x-coordinates by x_P and the y-coordinates by y_P. Then instead of P, Q and R, we can use the three points O(0, 0), A(x_A, y_A), B(x_B, x_B), shown in Fig. 3.5, where

$$x_A = x_Q - x_P \qquad x_B = x_R - x_P$$
$$y_A = y_Q - y_P \qquad y_B = y_R - y_P$$

We introduce the angle α between vector **a** (= **OA**) and the positive x-axis; similarly, β is the angle between **b** (= **OB**) and the positive x-axis (see Fig. 3.5).

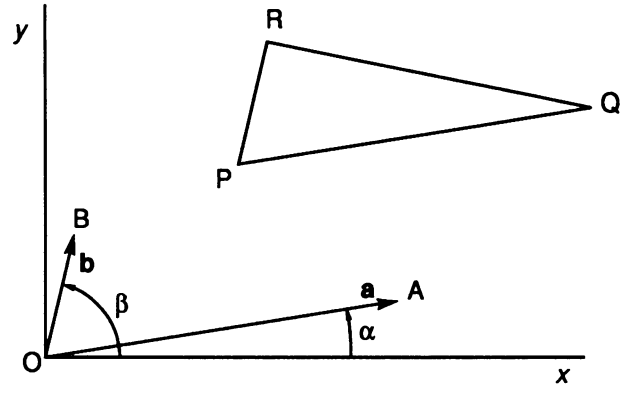


Fig. 3.5. Orientation of (P, Q, R) has the same sign as sin (β–α)

Then we can easily see that the orientation of OAB depends on the angle $\beta - \alpha$. If this angle lies between 0 and π, the orientation is clearly positive, but it is negative if this angle lies between π and 2π (or between $-\pi$ and 0). We can express this in an elegant and useful way by saying that the orientation in question depends on the value of $\sin(\beta - \alpha)$ rather than on the angle $\beta - \alpha$ itself. More specifically, the orientation has the same sign as

$$\sin(\beta - \alpha) = \sin\beta\cos\alpha - \cos\beta\sin\alpha = \begin{vmatrix} \cos\alpha & \sin\alpha \\ \cos\beta & \sin\beta \end{vmatrix} = D_0$$

Since we have

$$\cos\alpha = x_A/|\mathbf{a}| \qquad \sin\alpha = y_A/|\mathbf{a}|$$
$$\cos\beta = x_B/|\mathbf{b}| \qquad \sin\beta = y_B/|\mathbf{b}|$$

and because the lengths $|\mathbf{a}|$ and $|\mathbf{b}|$ are positive constants, we find the same sign if, instead of D_0, we compute

$$D = |\mathbf{a}|\,|\mathbf{b}|\,D_0 = \begin{vmatrix} x_A & y_A \\ x_B & y_B \end{vmatrix}$$

We have derived the coordinates of A and B from those of P, Q and R, so we can use D if the orientation of (P, Q, R) rather than that of (O, A, B) is desired. However, it is also possible to express the value of D directly in the coordinates of the points P, Q and R:

$$D = \begin{vmatrix} x_P & y_P & 1 \\ x_Q & y_Q & 1 \\ x_R & y_R & 1 \end{vmatrix}$$

The equivalence of these two expressions for D can be verified by subtracting the first row of the latter determinant from its second and its third rows.

Summarizing, we can compute D as either of the above two determinants, to find the orientation of (P, Q, R):

If $D > 0$, the points P, Q and R, in that order, are traversed counter-clockwise.
If $D = 0$, the points P, Q and R lie on a straight line.
If $D < 0$, the points P, Q and R, in that order, are traversed clockwise.

Testing whether a point lies inside a triangle

Determining the orientation of three points as we have just been discussing is useful in a test to see if a given point P lies inside a triangle ABC. Using a function **orientation** with three possible values 1, 0, –1, for positive, zero and negative, respectively, we can write

orientation(A, B, P) = orientation(B, C, P) = orientation(C, A, P)

if and only if P lies inside triangle ABC, as Fig. 3.6 illustrates.

The following function **inside** (which calls the function **orientation**) shows how we can realize this in C++; note type **vec**, introduced in Section 2.2. The small positive constant *EPS* is used instead of 0, so that determinants are regarded as nonzero only if their absolute values are greater than *EPS*. Recall that the expression **determinant > EPS**, as used here in the return-statement, has the value 1 or 0, which is just what we want:

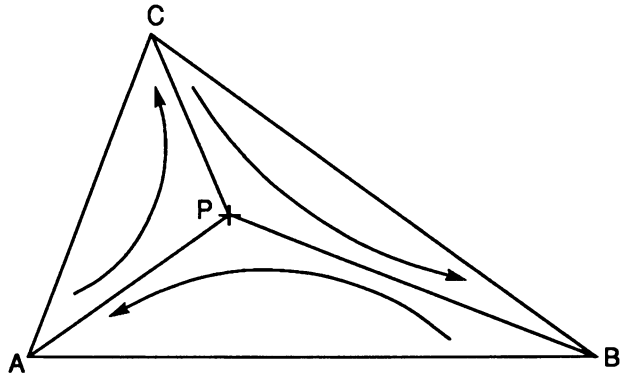


Fig. 3.6. Orientation used to test if P lies inside triangle ABC

```
int orientation(vec &P, vec &Q, vec &R)
{  const double EPS=1e-6;
   vec A, B;
   double determinant;
   A = Q - P; B = R - P;
   determinant = A.x * B.y - A.y * B.x;
   return (determinant < -EPS ? -1 : determinant > EPS);
}

int inside(vec &P, vec &A, vec &B, vec &C)
// Lies P inside triangle ABC?
{  int orABP = orientation(A, B, P);
   return (orientation(B, C, P) == orABP &&
           orientation(C, A, P) == orABP);
}
```

3.4 Vector Product

The *vector product* or *cross product* of two vectors **a** and **b** is written

$$\mathbf{a} \times \mathbf{b}$$

and is a vector **v** with the following properties. If $\mathbf{a} = c\mathbf{b}$ for some scalar c, then $\mathbf{v} = \mathbf{0}$. Otherwise the length of **v** is equal to

$$|\mathbf{v}| = |\mathbf{a}|\ |\mathbf{b}| \sin \gamma$$

where γ is the angle between **a** and **b**, and the direction of **v** is perpendicular to both **a** and **b** and is such that **a**, **b** and **v**, in that order, form a right-handed triple.[1] This means that if **a** is rotated through an angle $\gamma < 180°$ in the direction of **b**, then **v** has the direction of the advancement of a right-handed screw if turned in the same way. The following properties of vector products follow from this definition:

$$\begin{aligned} (k\mathbf{a}) \times \mathbf{b} &= k(\mathbf{a} \times \mathbf{b}) \qquad \text{for any real number } k \\ \mathbf{a} \times (\mathbf{b} + \mathbf{c}) &= \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c} \\ \mathbf{a} \times \mathbf{b} &= -\mathbf{b} \times \mathbf{a} \end{aligned}$$

In general $\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) \neq (\mathbf{a} \times \mathbf{b}) \times \mathbf{c}$. If we apply our definition of vector product to the unit vectors **i**, **j**, **k** (see Fig. 3.3), we have

---

[1] This definition assumes that we are using a right-handed coordinate system (see Section 3.1). Otherwise, we define $\mathbf{a} \times \mathbf{b} = \mathbf{v}$ such that **a**, **b** and **v**, like the unit vectors **i**, **j** and **k**, form a left-handed triple, so that we have $\mathbf{i} \times \mathbf{j} = \mathbf{k}$ with both types of coordinate systems.

$$\begin{array}{lll} \mathbf{i} \times \mathbf{i} = \mathbf{0} & \mathbf{j} \times \mathbf{j} = \mathbf{0} & \mathbf{k} \times \mathbf{k} = \mathbf{0} \\ \mathbf{i} \times \mathbf{j} = \mathbf{k} & \mathbf{j} \times \mathbf{k} = \mathbf{i} & \mathbf{k} \times \mathbf{i} = \mathbf{j} \\ \mathbf{j} \times \mathbf{i} = -\mathbf{k} & \mathbf{k} \times \mathbf{j} = -\mathbf{i} & \mathbf{i} \times \mathbf{k} = -\mathbf{j} \end{array}$$

Using these vector products in the expansion of

$$\mathbf{a} \times \mathbf{b} = (a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k}) \times (b_1\mathbf{i} + b_2\mathbf{j} + b_3\mathbf{k})$$

we obtain

$$\mathbf{a} \times \mathbf{b} = (a_2b_3 - a_3b_2)\mathbf{i} + (a_3b_1 - a_1b_3)\mathbf{j} + (a_1b_2 - a_2b_1)\mathbf{k}$$

which can be written as

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} \mathbf{i} + \begin{vmatrix} a_3 & a_1 \\ b_3 & b_1 \end{vmatrix} \mathbf{j} + \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix} \mathbf{k}$$

We rewrite this in a form that is very easy to remember:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_2 \end{vmatrix}$$

This is a mnemonic aid rather than a true determinant, since the elements of the first row are vectors instead of numbers.

If **a** and **b** are neighboring sides of a parallelogram, as shown in Fig. 3.7, the area of this parallelogram is the length of vector $\mathbf{a} \times \mathbf{b}$. This follows from our definition of vector product, according to which $|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \sin \gamma$ is the length of vector $\mathbf{a} \times \mathbf{b}$.

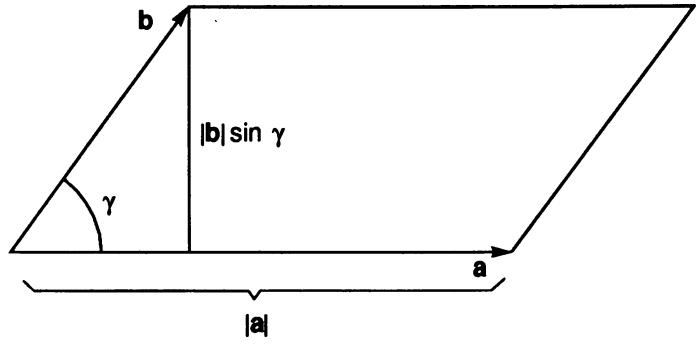


Fig. 3.7. Parallelogram with area $|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \sin \gamma$

3.5 Triangulation of Polygons

In Chapters 6 and 7 we will produce realistic pictures of three-dimensional objects whose boundary surfaces are polygons. This is no serious restriction since curved surfaces can be approximated by a great many polygons, in the same way as a curve is approximated by a sequence of line segments. Dealing with arbitrary polygons can lead to quite complex situations, especially if hidden line segments or surfaces are to be eliminated. Figure 3.8 shows an example of such a situation.

If all interior angles of a polygon are less than 180°, the polygon is said to be *convex*. In Fig. 3.9(b) the interior angles at vertices P_4 and P_5 are greater than 180°. We call such vertices *concave*. All other vertices in Figs. 3.9(a) and 3.9(b) are convex. If a polygon has at least one concave vertex, the polygon itself is said to be *non-convex*.

If A and B are two points on a convex polygon, the entire line segment AB belongs to the polygon. For a non-convex polygon this may not be the case. Non-convexity of polygons is a source of complexity and so is their variable number of vertices. For these reasons we pay special attention to triangles. These obviously have a fixed number of vertices and they are always convex. They are also interesting in connection with arbitrary polygons because any polygon can be divided into a finite number of triangles, and this section is about how this can be done. Division of a *convex* polygon into triangles is extremely simple, as Fig. 3.9(a) shows. If the vertices are successively numbered P_0, P_1,..., P_{n-1}, then drawing the diagonals P_0P_2, P_0P_3,..., P_0P_{n-2} is all that is needed. However, in a non-convex polygon, such as in Fig. 3.9(b), this simple method does not work, because some of the diagonals P_0P_2, P_0P_3,..., P_0P_{n-2} may not lie completely inside the polygon.

We will now discuss a general program module, TRIANGUL, which performs such a *triangulation* of a given polygon. We will use this module both here in a demonstration program, POLYTRIA, and in Chapters 6 and 7.

At first sight, it seems that a function (which we will call **triangul**) can perform this task only if we supply it with the coordinates of all polygon vertices. Doing this would not make this function as general as we would like, since it would deprive us

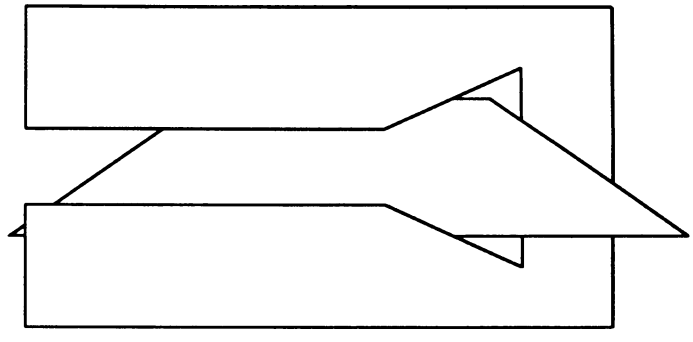


Fig. 3.8. Two polygons partly hiding each other

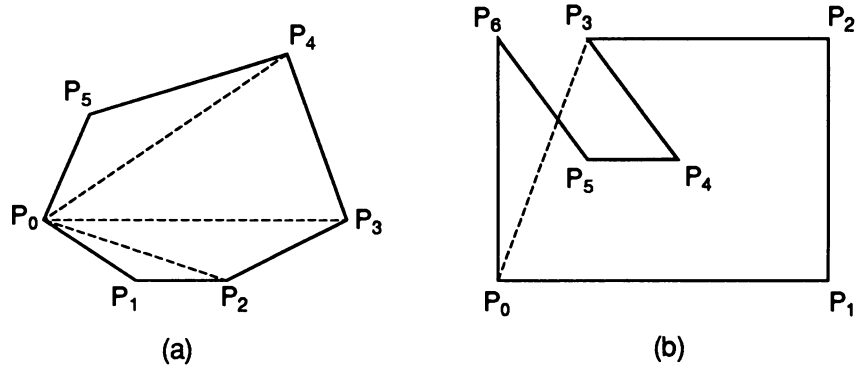


Fig. 3.9. (a) *Convex polygon*; (b) *non-convex polygon*

of the freedom to represent these coordinates in a way that depends on the application. For example, these coordinates are floating-point numbers in program POLYTRIA listed at the end of this section, while, for reasons of efficiency, they are integers in two more interesting programs, HIDELINE and HIDEFACE, in Chapters 6 and 7. Converting integers to floating-point numbers would seriously slow down those programs.

Fortunately, it is possible to write only one triangulation module, which, even without recompiling, can be used in all cases mentioned. This is because function **triangul** can do its work solely on the basis of the *orientation* of vertices. The user of module TRIANGUL must therefore supply a function **orienta**, which determines the orientation of any three vertices P, Q and R. Curiously enough, only this function **orienta** needs to use the coordinates of the vertices; all communication between the application module and function **triangul** is done by parameter passing, and this does not involve any coordinates, as the following header file shows:

```
// TRIANGUL.H: Header file for triangulation of polygons
struct trianrs {int A, B, C;};
int triangul(int *pol, int n, trianrs *nrs,
             int orienta(int P, int Q, int R));
```

The vertex numbers of the polygon, in counter-clockwise order, are given as **pol[0]**, **pol[1]**,..., **pol[n–1]**. The resulting triangles can be found in **nrs[0]**, **nrs[1]**,..., and the return value indicates how many triangles there are. In most cases there will be $n - 2$. For example, a square gives two triangles. However, there may be less than $n - 2$ triangles in special cases, as we will see at the end of this section.

Note that we find the same vertex numbers in **pol**, in **nrs**, and in argument lists that we use when calling function **orienta**. Consider, for example, polygon $P_0P_1P_2P_3$ of Fig. 3.10. Since this may be one of the faces of a polyhedron, we must not assume this polygon to have vertex numbers 0, 1, 2, 3. For example, vertices P_0, P_1, P_2 and P_3 may actually have numbers 123, 85, 27 and 100. This explains array **pol**, in which

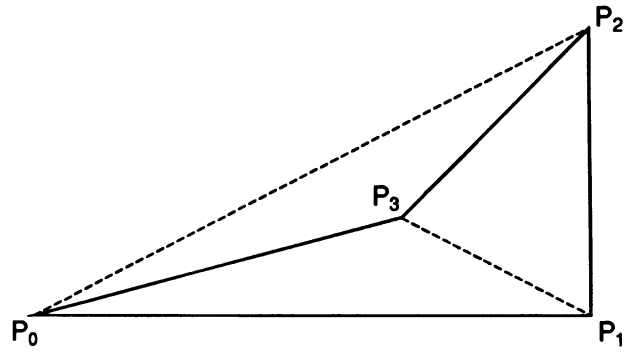


Fig. 3.10. Two types of diagonals

we store such vertex numbers for the polygon under consideration. In this example, the data supplied to function **triangul** would be

pol[0] = 123, **pol[1]** = 85, **pol[2]** = 27, **pol[3]** = 100
n = 4

together with function **orienta**, which, in the case of Fig. 3.10, must be written such that

orienta(pol[0], pol[1], pol[2]) = +1
orienta(pol[1], pol[2], pol[3]) = +1
orienta(pol[2], pol[3], pol[0]) = −1

Then function **triangul** will return the value 2, because there are two resulting triangles, and these are delivered by means of parameter **nrs**:

nrs[0].A = **pol[1]** **nrs[0].B** = **pol[2]** **nrs[0].C** = **pol[3]**
nrs[1].A = **pol[1]** **nrs[1].B** = **pol[3]** **nrs[1].C** = **pol[0]**

Reading P_0 instead of **pol[0]** and so on and consulting Fig. 3.10, we can clearly see that in this way array **nrs** represents the two resulting triangles. Note that the vertex numbers are in counter-clockwise order, both for the given polygon (in **pol**) and for the resulting triangles (in **nrs**).

Function **triangul** selects a triangle, of which two sides are successive edges of the polygon while the third is a diagonal. This triangle is then cut off the polygon, after which the process is repeated, until there is only one triangle left. If we follow the vertices of the polygon counter-clockwise, we cannot use three successive vertices if these are not counter-clockwise. For example, the triangle $P_2P_3P_0$ in Fig. 3.10 cannot be used because P_2, P_3 and P_0, in that order, are clockwise. The requirement for three successive vertices of the polygon to be counter-clockwise is necessary, but it is not sufficient. For example, although P_5, P_0 and P_1 in Fig. 3.11 are three successive vertices of the polygon and they are counter-clockwise, we cannot cut off triangle

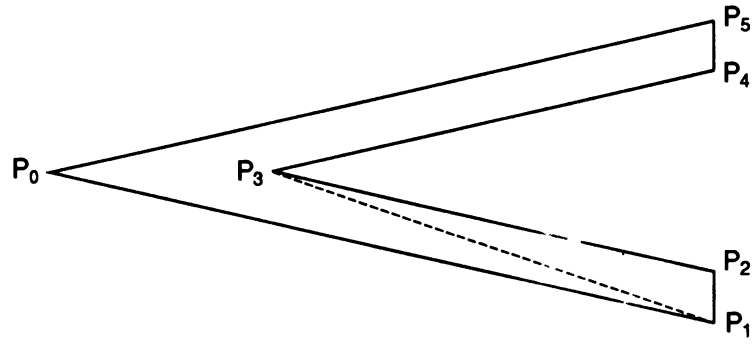


Fig. 3.11. Triangle $P_5P_0P_1$ cannot be used

$P_5P_0P_1$, because vertex P_3 lies inside it. Apparently, for each triangle that is a candidate to be cut off, we must check whether there are no other vertices lying inside it. As we have seen in Section 3.3, this too can be done by using the orientation concept. (Note that Fig. 3.11 demonstrates that it would not be safe to use the shortest diagonal that, together with two polygon edges, gives a triangle, since although P_1P_5 is shorter than P_1P_3, the latter diagonal is the preferred one.)

Function **triangul** uses a circular linked list, which initially has a node for each vertex of the polygon. During the triangulation process this list shrinks in the same way as the polygon, as Fig. 3.12 illustrates. Deleting an arbitrary element from a linked list can be done more efficiently than removing an array element and shifting all subsequent elements one position to fill the gap. We need such deletions each time we find a triangle that we can cut off the polygon.

When dealing with linked lists, we normally allocate and de-allocate memory for each individual node. Here we can do this for all n nodes together, which is faster. We will use pointer **ptr**, pointing to a block of n integers. Each value **ptr[i]** is such that **pol[i]** and **pol[ptr[i]]** are the numbers of two successive vertices of the remaining polygon. We use a 'start pointer' **q** and set up the cicular list as follows:

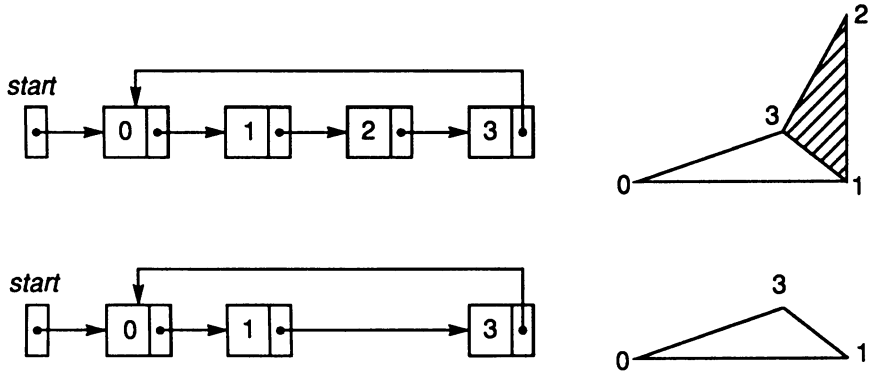


*Fig. 3.12. Data structures used in function **triangul***

q = 0, **ptr[0]** = 1, **ptr[1]** = 2,..., **ptr[n–2]** = **n–1**, **ptr[n–1]** = 0.

As Fig. 3.12 illustrates, cutting off a triangle corresponds to deleting a node from the circular list. In this example we have $n = 4$, and for simplicity, **pol[i]** = i for $i = 0$, 1, 2, 3. Then vertex 2 is deleted simply by setting **ptr[1]** = 3.

All this applies only to nonconvex polygons. In the important special case of *convex* polygons no linked list is set up and the triangles are obtained in the easiest possible way.

Program module TRIANGUL shows some more interesting programming details. After compiling this module in the form given below (also using the header file TRIANGUL.H which we have already seen), the resulting object module can be linked together with application modules, as we will see.

```
/* TRIANGUL.CPP: Triangulation of polygons
      Triangulation of a polygon with successive vertex numbers
      pol[0],..., pol[n-1], in counter-clockwise order.
      With three given vertex numbers P, Q, R, function
      orienta must determine their orientation:
      Negative = clockwise
      Zero     = collinear
      Positive = counter-clockwise
      If triangulation is possible, the resulting triangles are
      successively stored in array 'nrs'. Triangle j has vertex
      numbers nrs[j].A, nrs[j].B, nrs[j].C.
      Memory space for array 'nrs' must be supplied by the caller.
      Return value: the number of triangles found, or
                    -1 if no proper polygon or vertices clockwise.
                    -2: not enough memory.
*/
#include "triangul.h"

int triangul(int *pol, int n, trianrs *nrs,
             int orienta(int P, int Q, int R))
{   int *ptr, q, qA, qB, qC, r=-1, // -1 used as 'NULL'
        i, i1, i2, j, k, m, ok, ortB, *ort, polconvex=1,
        A, B, C, P, collinear;
    if (n < 3) return -1; // No polygon
    if (n == 3)
    {  nrs[0].A = pol[0]; nrs[0].B = pol[1]; nrs[0].C = pol[2];
       return 1;          // Only one triangle
    }
    ort = new int[n]; // ort[i] = 1 if vertex i is convex
    if (!ort) return -2;
    do
    {  collinear = 0;
       for (i=0; i<n; i++)
```

```
      {  i1 = (i < n - 1 ? i + 1 : 0);
         i2 = (i1 < n - 1 ? i1 + 1 : 0);
         ort[i1] = orienta(pol[i], pol[i1], pol[i2]);
         if (ort[i1] == 0)
         {  collinear = 1;
            for (j=i1; j<n-1; j++) pol[j] = pol[j+1];
            n--; break;
         }
         if (ort[i1] < 1) polconvex = 0;
      }
   }  while (collinear);
   if (n < 3) return -1;
   if (polconvex)  // Use diagonals through vertex 0:
   {  for (j=0; j<n-2; j++)
      {  nrs[j].A = pol[0];
         nrs[j].B = pol[j+1];
         nrs[j].C = pol[j+2];
      }
      delete[] ort; return n-2;
   }
   ptr = new int[n]; // ptr[i] is i's successor in linked list
   if (!ptr) return -2;
   // Set up a circular linked list of vertex numbers:
   for (i=1; i<n; i++) ptr[i-1] = i;
   ptr[n-1] = 0;
   q = 0; qA = ptr[q]; qB = ptr[qA]; qC = ptr[qB];
   j = 0;      // j triangles stored so far
   for (m=n; m>2; m--)  // m remaining nodes in circular list.
   {  for (k=0; k<m; k++)
      {  // Try triangle ABC:
         ortB = ort[qB]; ok = 0;
         // B is a candidate if it is convex:
         if (ortB > 0)
         {  A = pol[qA]; B = pol[qB]; C = pol[qC];
            ok = 1; r = ptr[qC];
            while (r != qA && ok)
            {  P = pol[r]; // ABC counter-clockwise:
               ok = P == A || P == B || P == C ||
               orienta(A, B, P) < 0 ||
               orienta(B, C, P) < 0 ||
               orienta(C, A, P) < 0;
               r = ptr[r];
            }
            // ok means: P coinciding with A, B, or C
            //           or outside ABC
            if (ok)
```

```
            {  nrs[j].A = pol[qA];
               nrs[j].B = pol[qB];
               nrs[j++].C = pol[qC];
            }
         }
         if (ok || ortB == 0)
         {  // Cut off triangle ABC from polygon:
            ptr[qA] = qC; qB = qC; qC = ptr[qC];
            if (ort[qA] < 1)
               ort[qA] = orienta(pol[q], pol[qA], pol[qB]);
            if (ort[qB] < 1)
               ort[qB] = orienta(pol[qA], pol[qB], pol[qC]);
            while (ort[qA] == 0 && m > 2)
            {  ptr[q] = qB; qA = qB;
               qB = qC; qC = ptr[qC]; m--;
            }
            while (ort[qB] == 0 && m > 2)
            {  ptr[qA] = qC; qB = qC; qC = ptr[qC]; m--;
            }
            break;
         }
         q = qA; qA = qB; qB = qC; qC = ptr[qC];
      }
   }
   delete[] ptr; delete[] ort; return j; // j triangles
}
```

We will now see how we can use this module TRIANGUL in a program that reads the coordinates of an arbitrary polygon's vertices from an input file and divides that polygon into triangles. For a polygon with n vertices the number n is given first, followed by the n coordinate pairs of successive vertices in counter-clockwise order. The output will be a drawing of the polygon, along with the resulting triangles. Actually, the triangles will be drawn slightly smaller than their real size, so that we can clearly see all resulting triangles, rather than the diagonals that form them.

The name of the input file can be supplied as a program argument, so we can start this program by entering the command

```
polytria polygon.txt
```

if POLYGON.TXT is the input file. If we do not supply the name of this file as a program argument, the program requests for input data to be entered on the keyboard in the same format as an input file would have.

The name of a graphics output file (in HP-GL format) may be given as a second program argument. For example, we can enter the command

```
polytria polygon.txt triangles.hpg
```

Note that the program will produce such an output file only if the input file is also supplied as a program argument. If there is only one program argument, this is regarded as an input file and there will be no graphics output file. Nor will there be one if we do not use program arguments at all (and enter all input data on the keyboard):

```
/* POLYTRIA.CPP: Dividing a polygon into triangles.
      To be linked with GRSYS, VEC and TRIANGUL
*/
#include <fstream.h>
#include <math.h>
#include "grsys.h"
#include "vec.h"
#include "triangul.h"
const float EPS=1e-6;

vec *p;        // n vertices p[0],..., p[n-1] are given.

void drawpolygon(vec *p, int n)
{  move(p[n-1]);
   for (int i=0; i<n; i++) draw(p[i]);
}

void drawtriangles(vec *p, trianrs *nrs, int m)
{  vec Centroid;
   vec A, B, C, A1, B1, C1;
   for (int j=0; j<m; j++)
   {  A = p[nrs[j].A];
      B = p[nrs[j].B];
      C = p[nrs[j].C];
      Centroid = 0.333333 * (A + B + C);
      A1 = Centroid + 0.90 * (A - Centroid);
      B1 = Centroid + 0.90 * (B - Centroid);
      C1 = Centroid + 0.90 * (C - Centroid);
      move(A1); draw(B1);
      draw(C1); draw(A1);
   }
}

int orienta(int Pnr, int Qnr, int Rnr)
{  vec A=p[Qnr]-p[Pnr], B=p[Rnr]-p[Pnr];
   float determinant;
   determinant = A.x * B.y - A.y * B.x;
   return (determinant < -EPS ? -1 :
           determinant > EPS);
}
```

```
int main(int argc, char *argv[])
{  int *nrspol;
   // Vertex number of triangles in nrspol[0],...,nrspol[n-1]
   trianrs *nrs; // Vertex numbers of triangle j in nrs[j]
   int n, i, ntria;
   ifstream inpfil;
   if (argc < 2)
   {  do
      {  cout <<
         "Enter n, followed by the coordinate pairs (x, y)\n"
         "of n vertices, in counter-clockwise order:\n";
         cin >> n;
      } while (n < 3);
   }  else
   {  inpfil.open(argv[1]);
      inpfil >> n;
   }
   p = new vec[n];
   nrspol = new int[n];
   nrs = new trianrs[n-2]; // At most n-2 triangles
   if (!p || !nrspol || !nrs)
      errmess("Not enough memory");
   for (i=0; i<n; i++)
   {  nrspol[i] = i;
      if (argc < 2) cin >> p[i].x >> p[i].y;
            else inpfil >> p[i].x >> p[i].y;
   }
   ntria = triangul(nrspol, n, nrs, orienta);
   if (ntria == -1) errmess("Polygon specification incorrect");
   if (ntria == -2) errmess("Not enough memory");
   if (argc > 2) initgr(argv[2]); else initgr();
   drawpolygon(p, n);
   drawtriangles(p, nrs, ntria);
   endgr();
   cout << n << " vertices; " << ntria << " triangles\n";
   return 0;
}
```

So far, we have used only very simple polygons as examples. The following set of input data describes a more complex one. The result of program POLYTRIA, executed with the above input file, is shown in Fig. 3.13. It shows both the original polygon (with only horizontal and vertical sides) and the resulting set of triangles, each drawn slightly smaller than its real size. Note that the number of triangles is 24, so we see that this number may be less than n–2, as mentioned in our discussion of TRIANGUL. It will be clear that the option of supplying the coordinates in a file is very convenient for a complex polygon such as this one:

```
32
1 1   4 1   4 2   5 2
5 1   9 1   9 6   5 6
5 5   4 5   4 6   1 6
1 4   2 4   2 5   3 5
3 4   6 4   6 5   8 5
8 4   7 4   7 3   8 3
8 2   6 2   6 3   3 3
3 2   2 2   2 3   1 3
```

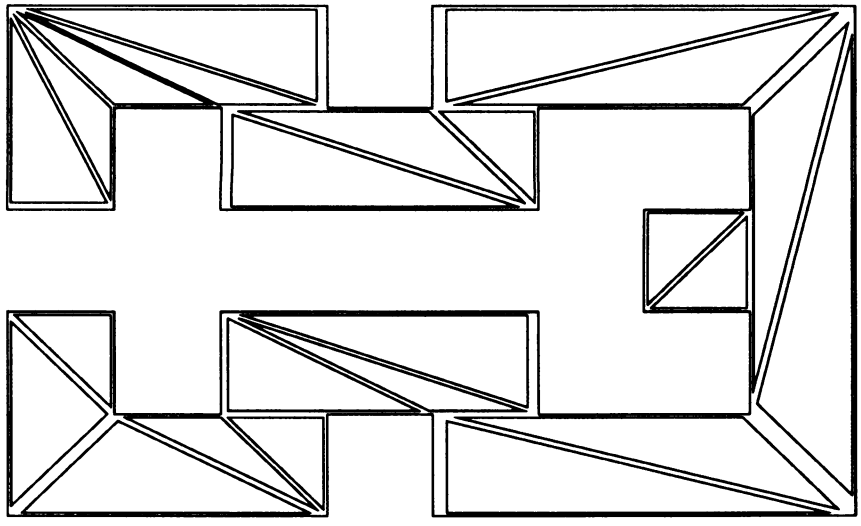


Fig. 3.13. Triangulation of a complex polygon with 32 vertices

3.6 Three-dimensional Rotations

As we have seen in Chapter 2, rotations in the xy-plane are relatively simple. The matrix notation used for them in Section 2.3 was not really necessary because we had already dealt with rotations without it in Section 2.1. In three-dimensional space (or *3-space*, for short) matrix notation is really very useful to perform arbitrary rotations, which we will discuss in this section.

Let us begin with the much simpler subject of *translations*, which, for 3-space, we can write as

$$x' = x + a_1$$
$$y' = y + a_2$$
$$z' = z + a_3$$

In the framework of the present subject, rotations, we prefer to write this as a matrix multiplication:

$$[x' \;\; y' \;\; z' \;\; 1] = [x \;\; y \;\; z \;\; 1]T$$

where

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a_1 & a_2 & a_3 & 1 \end{bmatrix} \tag{3.6}$$

As in Section 2.3, we need homogeneous coordinates here (with a 4 × 4 matrix) because the image $O' = (a_1, a_2, a_3)$ of the origin differs from O itself. With *rotations*, on the other hand, we have $O' = O$, as long as we rotate *about one of the coordinate axes*, and we do not need homogeneous coordinates for such special rotations, which we will discuss first. We will use a right-handed coordinate system, and we will call a rotation about a coordinate axis positive if it corresponds to the positive direction of that axis in the sense of a right-handed screw, as shown in Fig. 3.14.

Let us first rotate about the z-axis through an angle α. Using the abbreviations $c = \cos\alpha$ and $s = \sin\alpha$, we can write this rotation as

$$[x' \;\; y' \;\; z'] = [x \;\; y \;\; z]R_z$$

$$R_z = \begin{bmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

which follows from Section 2.3, Eq. (2.4).

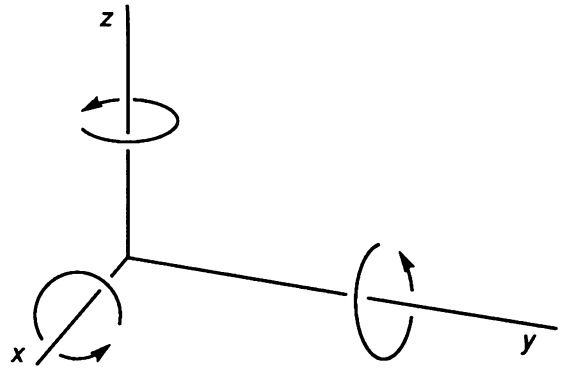


Fig. 3.14. Positive rotations about coordinate axes

This matrix R_z can be used to derive the matrices R_x and R_y for rotations about the other two axes in a formal way. We do this by means of the cyclic permutation obtained by replacing each of the letters x, y, z with its successor, the successor of z being x. We convert R_z to R_x by cyclic advancing each row one position, followed by similar operations on the columns:

$$R_z = \begin{bmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 1 \\ c & s & 0 \\ -s & c & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & s \\ 0 & -s & c \end{bmatrix} = R_x$$

Matrix R_x is converted to its successor R_y in the same way:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & s \\ 0 & -s & c \end{bmatrix} \rightarrow \begin{bmatrix} 0 & -s & c \\ 1 & 0 & 0 \\ 0 & c & s \end{bmatrix} \rightarrow \begin{bmatrix} c & 0 & -s \\ 0 & 1 & 0 \\ s & 0 & c \end{bmatrix} = R_y$$

Summarizing, we have the following matrices for rotation about the coordinate axes through an angle α:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix} \tag{3.7}$$

$$R_y = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix} \tag{3.8}$$

$$R_z = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.9}$$

For a rotation about the x-axis through the angle α, matrix R_x is used in the following way:

$$[x' \;\; y' \;\; z'] = [x \;\; y \;\; z]R_x$$

Matrices R_y and R_z are used in the same way for rotations about the other coordinate axes.

As explained in Section 2.1, equations for a transformation of points can also be interpreted as a change of coordinates. For example, moving all points a certain distance to the right requires the same equations as moving the coordinate system the same amount to the left. It follows that we need inverted matrices if we want the coordinate system to move in the same sense as the points would be moved. Fortunately, the inverses of T, R_x, R_y and R_z (see Eqs. (3.6)–(3.9)) can be written down immediately:

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a_1 & -a_2 & -a_3 & 1 \end{bmatrix} \tag{3.10}$$

$$R_x^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \tag{3.11}$$

$$R_y^{-1} = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix} \tag{3.12}$$

$$R_z^{-1} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.13}$$

We will now turn to the more interesting subject of rotation about any given line through the origin O (and will then generalize this by lifting the restriction that this line should pass through O). Actually, the rotations will be done about a *vector* rather than about a line, so we can define such a rotation (through a positive angle α) to be *positive* if it corresponds to the vector direction according to a right-handed screw. Let us use vector v, which starts at O and has the endpoint (v_1, v_2, v_3). Then we want to use the *spherical coordinates* ρ, θ and φ of this endpoint, shown in Fig. 3.15.

The usual way of expressing the relationship between these spherical coordinates and the rectangular coordinates (also known as *Cartesian* coordinates) v_1, v_2 and v_3 is as follows:

$v_1 = \rho \sin\varphi \cos\theta$
$v_2 = \rho \sin\varphi \sin\theta$
$v_3 = \rho \cos\varphi$

However, what we need here is the inverse of this relationship; we are given v_1, v_2 and v_3 and we want to compute ρ, θ and φ. The standard function **atan2** enables us to do this in C(++) in a very simple way:

```
#include <math.h>
...
rho = sqrt(v1 * v1 + v2 * v2 + v3 * v3);
theta = atan2(v2, v1);
phi = acos(v3/rho);
```

Our strategy will now be to change the coordinate system such that v, the axis of rotation, will lie on the new positive z-axis. We will therefore perform some coordinate transformations rather than point transformations, which means that the inverse matrices just mentioned are required. We begin with a rotation of the x- and y-axes about the z-axis through the angle θ. Using matrix (3.13), we can write this as

$$[x' \;\; y' \;\; z'] = [x \;\; y \;\; z]R_z^{-1}$$

$$R_z^{-1} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

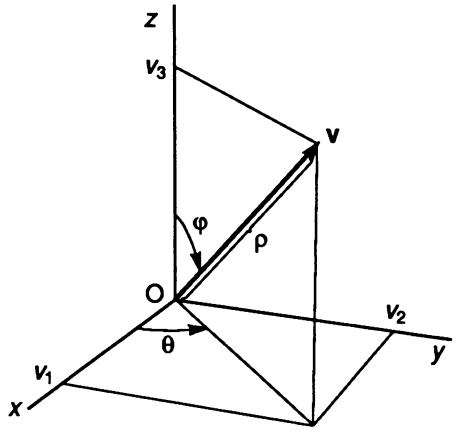


Fig. 3.15. Spherical coordinates

The positive x'-axis has the direction of vector $(v_1, v_2, 0)$. We then rotate the x'- and the z'-axes about the y'-axis, which, of course, is perpendicular to the x'-axis:

$$[x'' \;\; y'' \;\; z''] = [x' \;\; y' \;\; z']R_y^{-1}$$

$$R_y^{-1} = \begin{bmatrix} \cos\varphi & 0 & \sin\varphi \\ 0 & 1 & 0 \\ -\sin\varphi & 0 & \cos\varphi \end{bmatrix}$$

After these two coordinate transformations, we will now perform the actual rotation about vector v, which lies on the positive z''-axis. Referring to (3.9), we have

$$[x''' \;\; y''' \;\; z'''] = [x'' \;\; y'' \;\; z'']R_v$$

$$R_v = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The following matrix equation summarizes what we have achieved so far:

$$[x''' \;\; y''' \;\; z'''] = [x \;\; y \;\; z]R_z^{-1}R_y^{-1}R_v$$

Since we want the rotation about vector v to be expressed into the original coordinates x, y and z, we still have to revert from x''', y''' and z''' to these original coordinates by means of coordinate transformation; in other words, we have to use the inverses of the matrices R_z^{-1} and R_y^{-1}, in reverse order, after the rotation. The inverse matrices of R_z^{-1} and R_y^{-1} are R_z and R_y, so if we denote the coordinates of the rotated points by x^*, y^*, z^* (using the original coordinate system), we have

$$[x^* \;\; y^* \;\; z^*] = [x''' \;\; y''' \;\; z''']R_yR_z$$

This means that the complete rotation about vector v through the angle α is computed as

$$[x^* \;\; y^* \;\; z^*] = [x \;\; y \;\; z]R_z^{-1}R_y^{-1}R_vR_yR_z$$

where

$$R_z^{-1} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_y^{-1} = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

$$R_v = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Let us write R for the combined matrix that describes the rotation just mentioned:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = R_z^{-1} R_y^{-1} R_v R_y R_z \tag{3.14}$$

Up to now we have discussed a rotation about a vector v with origin O as its start point. We now want to lift this restriction. Instead, any point A(a_1, a_2, a_3) is to be allowed as a start point of vector v about which the rotation takes place. To achieve this, we still begin by computing matrix R of (3.14). We use this matrix in step (2) of the following three actions, which are required, in this order, for the desired rotation:

(1) Referring to (3.10), we perform a translation by moving the given point A to the origin O. This can be expressed by means of homogeneous coordinates and the following matrix:

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a_1 & -a_2 & -a_3 & 1 \end{bmatrix}$$

(2) We perform the rotation about a line through O as before, but we extend matrix R in a trivial way, so that we can use it for vectors in homogeneous coordinates:

$$R^* = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(3) We apply a translation opposite to that in (1), using the matrix

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a_1 & a_2 & a_3 & 1 \end{bmatrix}$$

The general rotation matrix is then

$$R_{GEN} = T^{-1}R^*T$$

and it is used as follows

$$[x^* \quad y^* \quad z^* \quad 1] = [x \quad y \quad z \quad 1]R_{GEN}$$

3.7 Vectors and Recursion

This section is about *Pythagoras' tree*, shown in Fig. 3.16. It is an example of a picture that can be generated in an elegant way by means of *recursion*. A function is said to be recursive if it conditionally calls itself. We will also use our VEC module for vectors, discussed in Section 2.2.

Let us begin by considering Fig. 3.17. If only the points A and B are given, we can compute the positions of the desired points C, D and E easily by using vectors. The triangles AFB and BGC are congruent. Using the origin O, we can define the vectors **a**, **b**, **c**, **d** and **e** as follows:

a = OA **b** = OB **c** = OC **d** = OD **e** = OE

Then we can use **a** and **b** to compute **c**, **d** and **e**:

$$\mathbf{c} = \mathbf{b} + \mathbf{BC} = \mathbf{b} + [x_C - x_B \quad y_C - y_B] = \mathbf{b} + [-(y_B - y_A) \quad x_B - x_A]$$
$$\mathbf{d} = \mathbf{a} + (\mathbf{c} - \mathbf{b})$$
$$\mathbf{e} = \mathbf{d} + 0.5(\mathbf{c} - \mathbf{a})$$

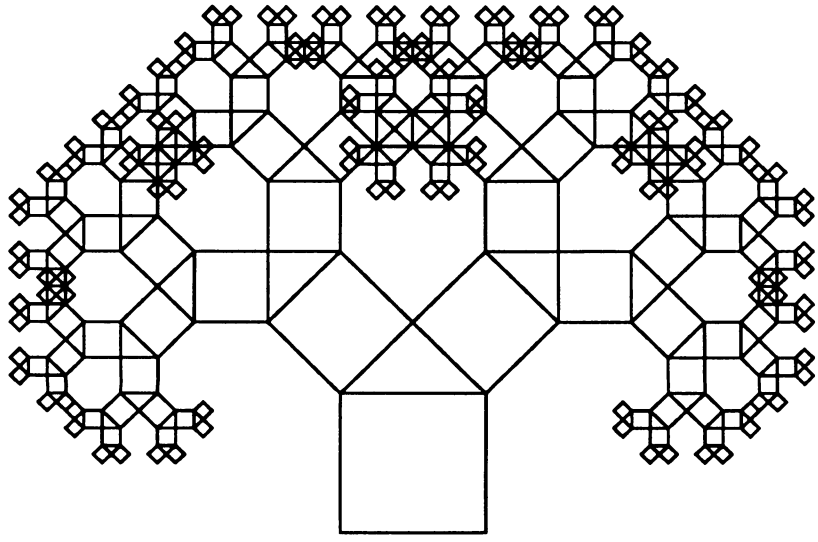


Fig. 3.16. Pythagoras' tree

Figure 3.16 was produced by program PYTHA. Function **Pythagoras** is given the two points A and B, so it can draw square ABCD. The points C and D, found as described above, are then also used in a recursive call, in the same way as A and B were used in the current call. The same applies to the points E and C, used in a second recursive call. Actually, these two recursive calls take place only if argument n is positive. The user of this program can enter the value of n (the *recursion depth*) that is used in the initial call to **Pythagoras** in the **main** function. The argument $n - 1$ is then used in recursive calls. The tree of Fig. 3.16 was obtained by using the following input data:

$x_A = 4.2$ $y_A = .3$ $x_B = 5.8$ $y_B = .3$ $n = 8$

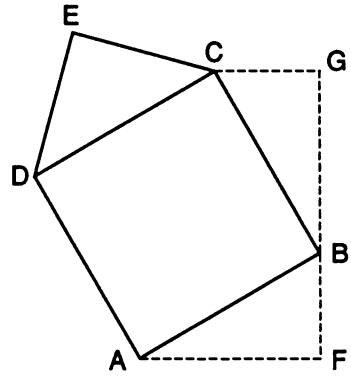


Fig. 3.17. Construction of points C, D and E

```
// PYTHA: Pythagoras' tree
//        To be linked together with the modules GRSYS and VEC
#include <iostream.h>
#include "grsys.h"
#include "vec.h"

void Pythagoras(vec &A, vec &B, int n)
{  if (n > 0)
   {  vec C = B + vec(A.y - B.y, B.x - A.x), D = A + C - B,
          E = D + 0.5 * (C - A);
      move(A); draw(B); draw(C); draw(D); draw(A);
      Pythagoras(D, E, n-1); Pythagoras(E, C, n-1);
   }
}

int main(int argc, char *argv[])
{  vec A, B;
   int n;
   cout << "Enter xA, yA, xB, yB and recursion depth n\n";
   cout << "(for example, 4.2  .3  5.8  .3  8): ";
   cin >> A.x >> A.y >> B.x >> B.y >> n;
   if (argc > 1) initgr(argv[1]); else initgr();
   // Optional program argument for output to HP-GL file
   Pythagoras(A, B, n);
   endgr(); return 0;
}
```

Exercises

3.1 Use the theory of Section 3.6 to write program module ROTA3D, which contains two functions, **initrotate** and **rotate**. The following header-file, which belongs to this module, shows the parameters of these functions:

```
// ROTA3D.H: Header file for 3D rotations
void initrotate(double a1, double a2, double a3,
   double v1, double v2, double v3, double alpha);
void rotate(double x, double y, double z,
   double *px1, double *py1, double *pz1);
```

This module is intended to be used for rotations about vector v = (v_1, v_2, v_3), with start point A(a_1, a_2, a_3), through the angle α. All these data are specified as arguments in a call to **initrotate**. This call initializes the rotation process; it computes the coefficients of matrix R_{GEN}, discussed at the end of Section 3.6. The actual rotation of points, mathematically expressed as

$$\mathbf{x}' = R_{GEN}\mathbf{x}$$

is then performed by calls to **rotate**. Use **static** variables for the coefficients just mentioned, declared globally in module ROTA3D. We will use this module in Exercise 8.5. In the meantime, you can demonstrate it by writing a simple application module, which first reads the arguments for **initrotate** from the keyboard; then it repeatedly reads the coordinates x, y and z of points P and displays the coordinates of the rotated points P′.

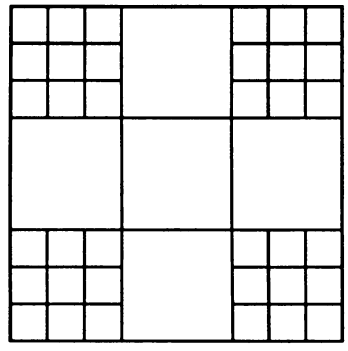


Fig. 3.18. Recursive subdivision of squares (Exercise 3.2)

3.2 Divide a square into nine squares of equal size, as illustrated by Fig. 3.18. Then the same is to be done for the four small squares at the corners of the original square, and so on. Use a recursive function, and a (maximum) recursion depth specified in the same way as in program PYTHA in Section 3.7.3

3.3 Write a program that reads the coordinates of the vertices of triangle ABC. Connect the centers of the three triangle sides, as shown in Fig. 3.19. Apply this principle recursively to the three smaller triangles of which A, B or C is a vertex; in other words, it is not applied to the small triangle in the center of the original one. Again, use a maximum recursion depth specified as input data.

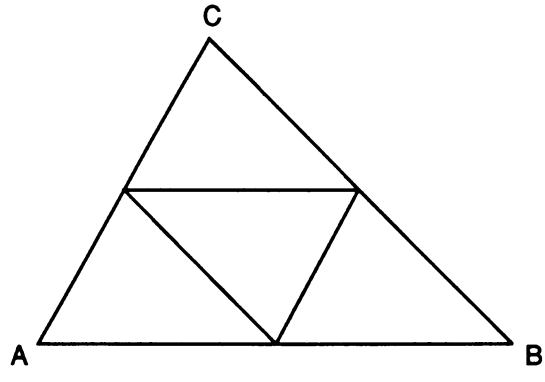


Fig. 3.19. Lines in triangle (Exercise 3.3)

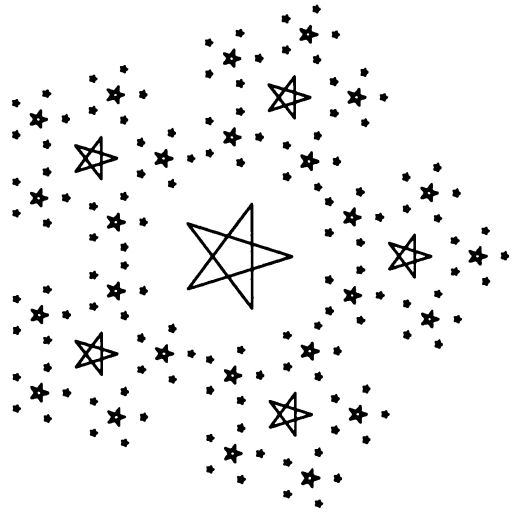


Fig. 3.20. Stars, produced by a recursive function (Exercise 3.4)

3.4 Write a recursive function to draw stars, surrounded by smaller stars (up to a given recursion depth) as shown in Fig. 3.20.

3.5 Write a program that draws the *circumscribed circle* (also known as the *circumcircle*) of a given triangle ABC; this circle passes through the points A, B and C. Remember, the three perpendicular bisectors of the three sides of a triangle all pass through one point, the *circumcenter*, which is the center of the circumscribed circle.

3.6 Write a program that reads the coordinates of the three points A, B and C and draws an circular arc, starting at A, passing through B and ending at C (see also Exercise 3.5).

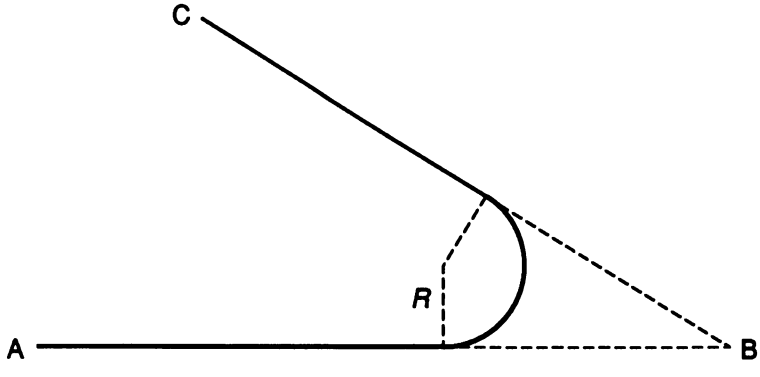


Fig. 3.21. Fillet (Exercise 3.7)

3.7 Construct a *fillet* to replace a sharp corner with a rounded one, as illustrated by the solid lines and the arc in Fig. 3.21. The three points A, B, C and the radius R are given.

3.8 A triangle ABC is given. Construct its *inscribed circle* (also known as its *incircle*). The center of this circle lies on the point of intersection of the (internal) bisectors of the tree angles A, B and C. Draw also the three *excircles*, which, like the incircle, are tangent to the sides of the triangle, as shown in Fig. 3.22. The centers of the excircles lie on the points of intersection of the external bisectors of the three angles A, B and C.

Use a transformation from a window to a viewport by means of uniform scaling, as discussed in Section 2.6. After computing the centers and the radii of the excircles, you can use these to find the window boundaries. Thanks to this transformation, everything will just fit into the screen boundaries, for any three given points A, B and C (not lying on the same straight line).

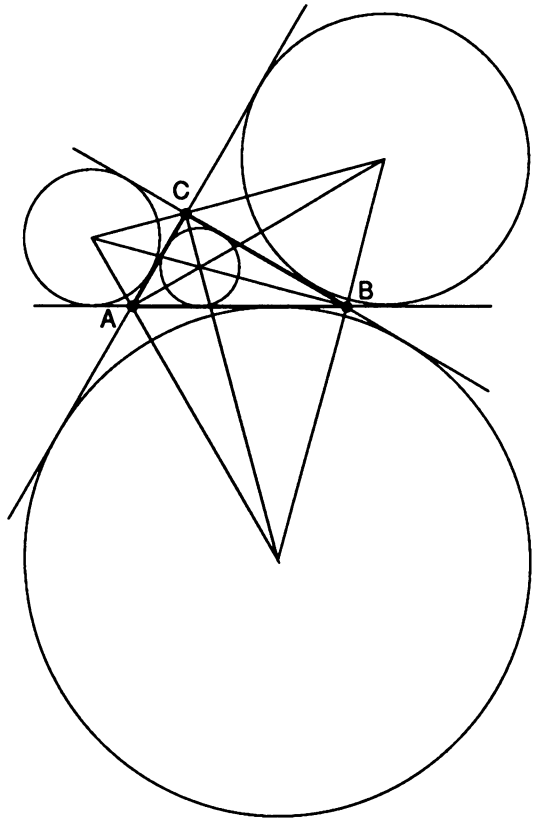


Fig. 3.22. Incircle and excircles of triangle ABC (Exercise 3.8)

4

Using Pixels

4.1 Pixels and Colors

Most video displays can be regarded as consisting of a great many tiny squares. Each such square is called a *pixel*[1], which stands for 'picture element'. For example, with standard VGA on the IBM PC the pixels are arranged in 480 rows and 640 columns, which gives 480 × 640 = 307200 pixels. In contrast to what we did in the previous chapters, we will now address the individual pixels.

Figure 4.1 shows a new type of coordinates, which are integers, denoted by X and Y. As is usual in mathematics, the origin lies at the bottom left, and we have

$X = 0, 1, 2, \ldots, X\_\_max$
$Y = 0, 1, 2, \ldots, Y\_\_max$

where $X\_\_max$ and $Y\_\_max$ are device independent. With the example just mentioned, we have

$X\_\_max = 639$
$Y\_\_max = 479$

With monochrome video displays, each pixel can be either dark or light, which we regard as two colors, coded 0 for dark and 1 for light. (There are also monochrome

[1] With some video displays, such as the Hercules Graphics Adapter for the IBM PC, each pixel is not a square but rather a rectangle, its height being greater than its width. We will ignore such displays, restricting our discussion to square pixels, which are used in newer display types, such as VGA.

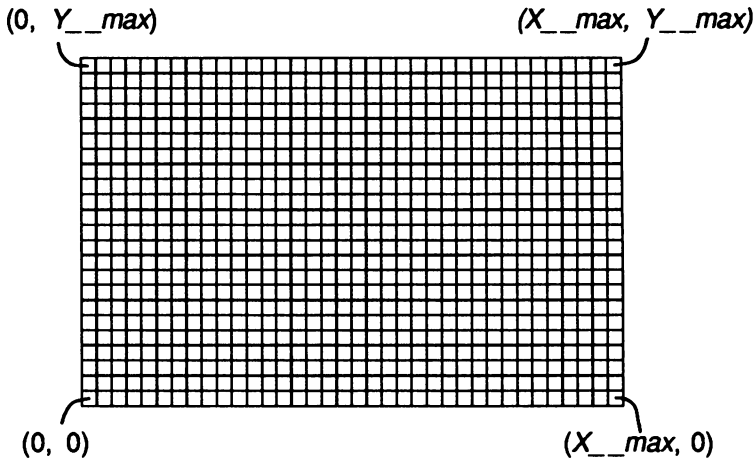


Fig. 4.1. Pixels

video displays with illumination varying in intensity; we will not use these in this book.) With color screens, on the other hand, there is a range of more than two colors.

In order to deal with pixels and colors, we need some more graphics primitives. These are listed below; they consist of both functions and global variables (similar to *x_max* and so on, used in the preceding chapters), and belong to module GRSYS.

Functions:

get_maxcolor( ) Returns the maximum integer color code that can be used. Function **initgr** calls this function and assigns its returned value plus 1 to the global variable *ncolors*.

set_color(color) Makes the current foreground color equal to argument *color*, which must be one of the values 0, 1,..., *ncolors* – 1. If we do not call this function, there is a default foreground color. This is different from the *background color*, which is the initial color of the entire screen.

set_backgroundcolor(color)
Makes the current background color equal to *color*.

putpix(X, Y) Assigns the current foreground color to pixel (*X*, *Y*), where *X* and *Y* are integers ranging from 0 to *X__max* and *Y__max*.

ix(x), iy(y) Functions to convert **float** coordinates (as used so far) to pixel coordinates.

Global variables (*defined only after a call to* **initgr**):

ncolors The number of possible colors. The available color codes range from 0 to *ncolors* – 1.

foregrcolor	The current foreground color.
backgrcolor	The current background color.
X__max	Maximum *X* coordinate (integer) as shown in Fig. 4.1.
Y__max	Maximum *Y* coordinate (integer) as shown in Fig. 4.1.

Here are some typical color values, which apply to the VGA 16-color mode on the IBM PC:

0	black	4	red	8	dark gray	12	light red
1	blue	5	magenta	9	light blue	13	light magenta
2	green	6	brown	10	light green	14	yellow
3	cyan	7	light gray	11	light cyan	15	white

In Chapter 7 we will discuss how to define colors ourselves by means of *palettes.*

Even if we have a color screen we can ignore colors and simply use the default value of *foregrcolor*. With this monochrome mode, there are two possibilities:

(1) The background color is dark (for example, black or blue) and the foreground color is light (white or yellow).
(2) The background color is light (white or gray) and the foreground color is black.

At first sight, (2) looks more logical than (1), because it corresponds to the way text and line drawings usually appear on paper. On the other hand, if it is desirable that most of the screen is dark, (1) may be a better solution. However, in this case the pictures on the screen must be inverted to obtain figures such as most illustrations in this book.

Program GCDPLOT demonstrates the use of the function **putpix**. It can best be explained by means of its output, shown in Fig. 4.2. Note that black is the foreground color in this printed version, so a black square in this figure may correspond to a white square on the screen if there is a dark background color.

Figure 4.2 shows a large square consisting of 15 rows and 15 columns, which gives 15 × 15 small black or white squares, each consisting of 25 × 25 pixels. Both the rows and the columns should be numbered from 2 to 16, starting at the bottom and on the left, respectively. Each square in Fig. 4.2 is black if and only if the greatest common divisor[1] (*GCD*) of its row and column numbers is equal to 1. For example, the square in the bottom-left corner is white because *GCD*(2, 2) = 2, but the one immediately above it is black because *GCD*(2, 3) = 1. As you can see, many black squares are found on rows and column that correspond to prime numbers, such as 11 and 13, while numbers with many divisors, such as 12, correspond to rows and columns with many white squares.

A much simpler version of program GCDPLOT is possible if we do not care about efficiency. However, when the same amount of work is to be done for many thousands

[1] The greatest common divisor of two integers is computed by using Euclid's algorithm, discussed in many books on programming, such as *Programs and Data Structures in C* by the same author.

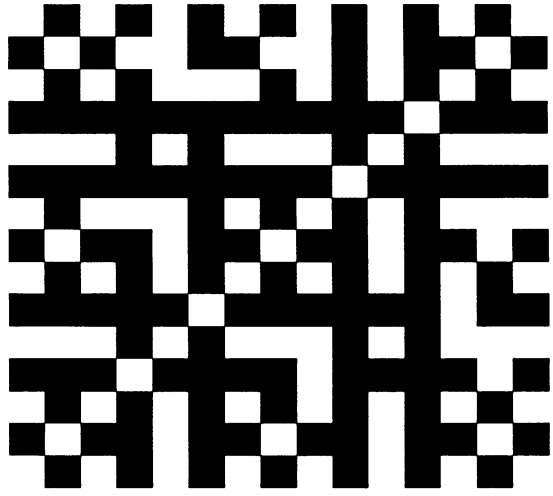


Fig. 4.2. Output of program GCDPLOT with k = 25

of pixels, we want this to be done as fast as possible. The version shown here does not therefore compute values inside a loop if these can be computed before the loop starts. Another complicating factor is that we want the picture to be in the middle of the screen. Finally, a very simple version of this program can be written if we let each black or white square consist of only one pixel. This is indeed reasonable if our video screen has a very low resolution. However, with 640 × 480 pixels, for example, this approach would lead to a picture in which the pattern would not be clear for normal eyesight. With our version, the number of pixels for the sides of the squares is read from the keyboard and assigned to the variable *k*. Figure 4.2 was produced with *k* = 25 (instead of *k* = 1 as just suggested).

```
/* GCDPLOT: Demonstration program dealing with pixels
            To be linked with module GRSYS.
*/
#include <iostream.h>
#include <stdlib.h>
#include "grsys.h"

int gcd(int a, int b)
{  int r;
   while (b){r = a % b; a = b; b = r;}
   return a;
}

int main()
{  int k, x, y, xmargin, ymargin, n, N, x1, y1, xplus2,
       X, i, j;
```

```
    cout << "Dimension of elementary square (e.g. 25): ";
    cin >> k; // Each small square will consist of k x k pixels
    initgr();              // 100 pixels in margins (vertically):
    n = (Y__max - 100)/k; // n x n small squares
    N = n * k;             // N x N pixels in large square
    xmargin = (X__max - N)/2;  // Left margin
    ymargin = (Y__max - N)/2;  // Bottom margin
    for (x=0; x<n; x++)
    {  x1 = xmargin + x * k; xplus2 = x + 2;
       for (y=0; y<n; y++)              // Square (x, y)
       {  if (gcd(xplus2, y+2) == 1)  // Start with gcd(2, 2)
          {  y1 = ymargin + y * k;
             for (i=0; i<k; i++)       // Pixel: X = xmargin+x*k+i
             {  X = x1 + i;            //        Y = ymargin+y*k+j
                for (j=0; j<k; j++) putpix(X, y1 + j);
             }
          }
       }
    }
    endgr(); return 0;
}
```

4.2 Line Drawing by Writing Pixels

In the preceding chapters we have taken for granted that lines can be drawn by our functions **move** and **draw**. These high-level functions are based on floating-point coordinates, which we will now call *real coordinates*. At a lower level, they must be converted to integer *pixel coordinates*, discussed in the previous section. Since the total width of *X__max* pixels is equal to *x_max* – *x_min*, the width of one pixel is

$$\frac{x\_max - x\_min}{X\_\_max}$$

In the same way, we find

$$\frac{y\_max - y\_min}{Y\_\_max}$$

as the height of a pixel. Since we have assumed that our pixels are squares (see the footnote at the beginning of this chapter), these two expression have the same value. We will actually use the inverse of this value, which we may refer to as the *density*, since it is the number of pixels on a line segment of unit length. As we know, global program variables such as *x_max*, *X__max* and so on are defined in function **initgr**.

We now introduce the floating-point variable *density*, and insert the following statement in this function:

```
density = X__max / (x_max - x_min);
//  This is equal to Y__max / (y_max - y_min).
```

The conversion from real coordinates x and y to pixel coordinates X and Y is then done by the statements

```
X = int(density * (x - x_min));
Y = int(density * (y - y_min));
```

(Note that we obtain the correct values for X and Y if we substitute, for example, *x_min* or *x_max* for x). Since such conversions are often required, it is convenient that they can also be done by calling the two functions, **ix** and **iy**, declared in GRSYS.H. Instead of the above two statements we can simply write

```
X = ix(x);
Y = iy(y);
```

However, if speed is at stake, we should seriously consider the feasibility of not using floating-point coordinates at all.

Bresenham's algorithm for lines

In the rest of this chapter we will very often use pixel coordinates. If there is no danger of confusion, we may as well denote them by lower case letters x and y. Using capital letters X and Y all the time would be very unusual; in the rest of this chapter they will therefore be used only if these (integer) pixel coordinates are to be distinguished from real coordinates x and y.

Figure 4.3 shows a line segment with endpoints P(1, 1) and Q(12, 5) (where 1, 12 and 5 are integer pixel coordinates). In Section 4.1 we have introduced pixels as tiny *squares*, so you may wonder how these relate to *grid points* such as P and Q in Fig. 4.3. The answer is that these grid points are to be regarded as the *centers* of those tiny squares (and *not* as their corners).

Any straight line has to be approximated by a sequence of points. Its endpoints are given by pixel coordinates, so these need not be approximated. For example, the endpoints P(1, 1) and Q(12, 5) of the line in Fig. 4.3 are given, and it is our task to find the points (2, 1), (3, 2), (4, 2), (5, 2), (6, 3), (7, 3), (8, 4), (9, 4), (10, 4) and (11, 5) between P and Q, and to put all these pixels (including P and Q themselves) on the screen. We want this to be done by function **draw_line**, which is declared as follows:

```
void draw_line(int xP, int yP, int xQ, int yQ);
```

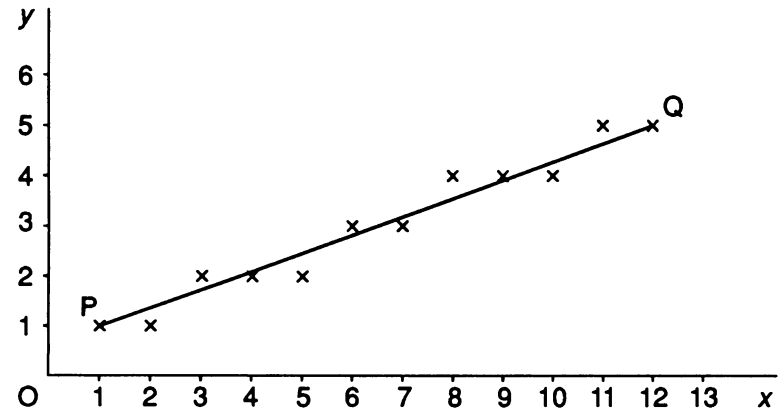


Fig. 4.3. Grid points approximating a line segment

Line PQ in Fig. 4.3 has an angle of inclination which is less than 45°; in other words, we have

$$|y_Q - y_P| \leq |x_Q - x_P|$$

In such cases we want to find exactly one point (x, y) for every x between x_P and x_Q; in other words, several of these points, such as (3, 2), (4, 2) and (5, 2) may lie on the same horizontal line, but no two such points may lie on the same vertical line.

The simplest way of finding the desired points between P and Q is by means of floating-point arithmetic, but for reasons of efficiency we prefer using only integers. In spite of this, we will first write a preliminary version, **draw_line1**, which uses some **float** variables. This is easier to understand than an integer version, which we will derive from it. This first version assumes that the coordinates of P and Q satisfy

$$\begin{aligned} &x_P < x_Q \\ &y_P \leq y_Q \\ &0 \leq y_Q - y_P \leq x_Q - x_P \end{aligned} \tag{4.1}$$

Writing θ for the angle of inclination, we can infer from this that this angle and the slope $m = \tan\theta$ satisfy

$$0 \leq \theta \leq \tfrac{1}{4}\pi$$

$$0 \leq m = \frac{y_Q - y_P}{x_Q - x_P} \leq 1 \tag{4.2}$$

Let us write x and y for the coordinates of grid points, which implies that they are integers, in contrast to y_{exact}, which is the y-coordinate of the corresponding point on

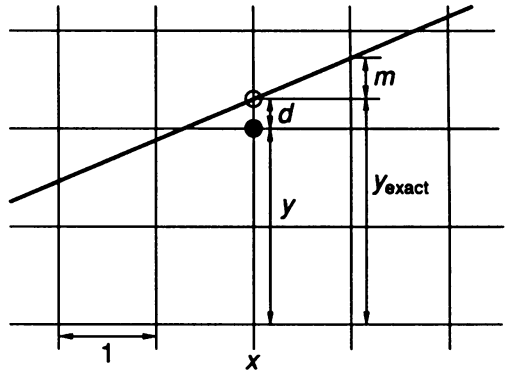


Fig. 4.4. Slope m and error d

line PQ, as shown in Fig. 4.4. Since we are using integer pixel coordinates, two successive x-values have difference 1 and the y-coordinates of the corresponding points on line PQ have difference m.

Figure 4.4 shows this interpretation of m as well as that of the 'error'

$$d = y_{exact} - y \tag{4.3}$$

Our function **draw_line1** is based on the fact that this error d must not be greater than 0.5, but we can choose the grid points such that

$$-0.5 < d \leq +0.5 \tag{4.4}$$

The following function **draw_line1** does not perform any multiplications or divisions within the while-loop:

```
void draw_line1(int xP, int yP, int xQ, int yQ)
{  int x=xP, y=yP;
   float d=0, m=float(yQ-yP)/float(xQ-xP);
   while (x != xQ)
   {  putpix(x, y);
      x++;
      d += m;
      if (d > 0.5){y++; d--;}
   }
}
```

This version is easy to understand if we pay attention to Fig. 4.4. The first call to **putpix** applies to point P, so we begin with error $d = 0$. In each step of the loop, x is increased by 1. Assuming for the moment that y will not change, it follows from (4.3) that the growth of d will be the same as that of y_{exact}, so d is to be increased by

m. However, this may result in d becoming greater than 0.5, which would violate (4.4). If this happens, the assumption that y can be left unchanged proves wrong; we then increase y by 1 and at the same time, in accordance with (4.3), we decrease d by 1, after which d still satisfies (4.4).

We use **draw_line1** to derive a much faster version, **draw_line2**, from it by replacing type **float** with type **int**. Although, in general, **float** variables are used for any real numbers, our **float** variable m is a special case in that it represents a rational number, that is, an integer numerator divided by an integer denominator, as (4.2) shows. Function **draw_line1** also shows that variable d is initially zero and is not altered other than by adding m to it and subtracting 1 from it. Thus variable d is a rational number with the same denominator as m. It is then a good idea to switch to integers, by using this denominator as a common multiplication factor. Besides the **float** variables m and d, **draw_line1** also uses the floating-point constant **0.5**. To eliminate this as well, we actually use the multiplication factor $c = 2(x_Q - x_P)$, and replace the real numbers m and d and 0.5 with the integers M, D and Δx, which are c times as large. Thus we have

$$M = cm = 2(y_Q - y_P)$$
$$D = cd$$
$$\Delta x = c \times 0.5 = x_Q - x_P$$

so that we can replace the program lines

```
d += m;
if (d > 0.5){y++; d--;}
```

with

```
D += M;
if (D > dx){y++; D -= c;}
```

Here is our improved function, which uses only integer arithmetic:

```
void draw_line2(int xP, int yP, int xQ, int yQ)
{  int x=xP, y=yP, D=0, dx=xQ-xP, c=2*dx, M=2*(yQ-yP);
   while (x != xQ)
   {  putpix(x, y);
      x++;
      D += M;
      if (D > dx){y++; D -= c;}
   }
}
```

Although function **draw_line2** is very fast, we will not use it in this form because it is restricted to the relative positions of P and Q given by (4.1). If the slope of line PQ is greater than 1, the independent variable (which in the loop is unconditionally increased by 1) must be y instead of x. Also, both for x and for y, we must find out

in advance whether these variables are to be increased or decreased by 1. All this is realized in our final, general function **draw_line**:

```
void draw_line(int xP, int yP, int xQ, int yQ)
{  int x=xP, y=yP, D=0, dx=xQ-xP, dy=yQ-yP, c, M,
       xinc=1, yinc=1;
   if (dx < 0){xinc = -1; dx = -dx;}
   if (dy < 0){yinc = -1; dy = -dy;}
   if (dy < dx)
   {  c = 2 * dx; M = 2 * dy;
      while (x != xQ)
      {  putpix(x, y);
         x += xinc; D += M;
         if (D > dx){y += yinc; D -= c;}
      }
   }  else
   {  c = 2 * dy; M = 2 * dx;
      while (y != yQ)
      {  putpix(x, y);
         y += yinc; D += M;
         if (D > dy){x += xinc; D -= c;}
      }
   }
}
```

The idea of drawing lines by means of only integer variables was first realized by Bresenham; his name is therefore associated with the algorithm used here, as it is with a similar algorithm for circles, to be discussed in the next section.

Horizontal lines

It may be worthwhile to write a special line-drawing function for horizontal lines. Thanks to the simplicity of this problem, this function will be faster than the above more general function **draw_line**:

```
void horline(int xleft, int xright, int y)
{  while (xleft <= xright) putpix(xleft++, y);
}
```

Since this function will turn out to be useful in Section 4.4, we will add it to the module GRSYS.

4.3 Circles

If speed is not a critical factor, the simplest way to draw a circle with a given center C and radius r is by connecting neighboring points (x, y) computed as

$$x = x_C + r \cos \varphi$$
$$y = y_C + r \sin \varphi$$

where

$$\varphi = i \times \frac{2\pi}{n} \qquad (i = 0, 1, 2, \ldots, n-1)$$

for some large value of n.

Instead of this rather slow way of drawing circles, based on floating-point arithmetic, there is a much faster method, which uses only integers. Let us begin with circles that have the origin O(0, 0) as their centers. The equation of such a circle with radius r is

$$x^2 + y^2 = r^2$$

Recall that in the previous section we started with a special case. We will do the same here, and consider first the arc PQ on the circle, where point P(r, 0) is the top of the circle and point Q lies on the line $y = x$, to the right of P. Thus this arc is the eighth part of the full circle, belonging to the second octant, as shown in Figs. 4.5 and 4.6.

With a given grid, Fig. 4.6 shows the pixels that approximate the arc of the second octant. If we can find these pixels, we can use them to find those of the other seven octants by symmetry.

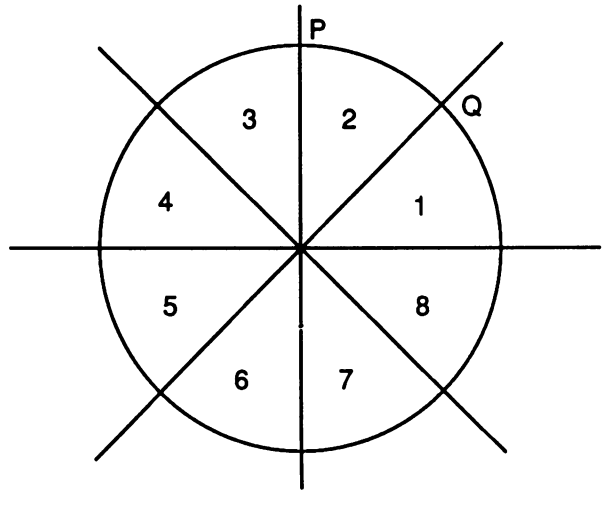


Fig. 4.5. The eight octants of a circle

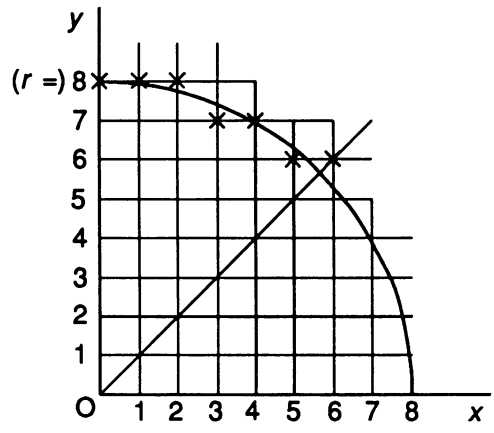


Fig. 4.6. The arc of octant 2, approximated by pixels

If we start at P and move to Q along the arc, the x-coordinate changes more quickly than the y-coordinate. In contrast to some other gridpoints that approximate the arc, start point P(0, r) lies on the circle. To find the next point, we increase x by 1, and we have to choose between the two points (1, r) and (1, r–1). The situation for subsequent points is similar: after finding point (x, y), we increase x and we have to choose between y and y–1. This suggests that, after increasing x, we compute the two sums

$$x^2 + y^2 \qquad \text{and} \qquad x^2 + (y-1)^2$$

to see which lies closer to r^2. If it is the first, we leave y unaltered; otherwise we decrease y by 1. Note that all this can be done by using only integer arithmetic, so in principle our problem is solved. However, this method is not the fastest possible because of two drawbacks. First, it requires some rather time-consuming multiplications and, second, the squares of x and y may be too large for type **int**, so that we would have to use type **long int**, which is slower than type **int**. Fortunately, squaring x and y is not really necessary. At start point P we have $x = 0$ and $y = r$, so $x^2 + y^2$ is exactly equal to r^2 at this point. Since x will successively assume the values

0, 1, 2, 3, 4,...

the corresponding values of x^2 will be

0, 1, 4, 9, 16,...

Note that we have

$$1 - 0 = 1$$
$$4 - 1 = 3$$
$$9 - 4 = 5$$
$$16 - 9 = 7$$

so the difference u between two squares increases by 2 each time. This observation enables us to compute the above squares without performing any multiplications. We can obtain the sequences

$$x = 0, 1, 2, 3, 4,...$$
$$x^2 = 0, 1, 4, 9, 16,...$$
$$u = 1, 3, 5, 7, 9,...$$

in the following, very efficient way:

```
x = x2 = 0; u = 1;
while (...)
{  x++;
   x2 += u;
   u += 2;
}
```

We still have large squares, so that type **long int** may be required for the variable $x2$. However, we are not really interested in these squares themselves, but rather in the values of the 'error'

$$E = x^2 + y^2 - r^2$$

the absolute value of which we want to be as small as possible. Since this value is initially 0, we can find all subsequent values of E if we know how E changes in each step. If we increase x by 1 and leave y unaltered, E increases by the same value as x^2. In the alternative case, we also change y, that is, we decrease y by 1. In that case we want to know how this influences E, so we need to know how much y^2 decreases if y is decreased by 1. Unlike x, the value of y does not change in each step, but if it does we can find the new value of y^2 without multiplying. The successive values of y^2 are

$$r^2, (r-1)^2, (r-2)^2,...$$

The differences v between two successive terms in this sequence are

$$r^2 - (r-1)^2 = 2r - 1$$
$$(r-1)^2 - (r-2)^2 = 2r - 3$$
$$(r-2)^2 - (r-1)^2 = 2r - 5$$
$$\vdots$$

Thus this sequence of v-values starts with $2r-1$ and decreases by 2 in each step. We now see that after increasing E unconditionally by u as a consequence of increasing x by 1, we may or may not decrease E by v, depending on whether or not y should decreased by 1. This means that (after increasing E by u) we must check if

$$|E - v| < |E| \tag{4.5}$$

If this is the case, the 'error' E can be diminished by decreasing E by v and y by 1.

For reasons of efficiency, it is useful to know that (4.5) is equivalent to the following inequality:

$$v < 2E \tag{4.6}$$

You can verify this by remembering that (4.5) is equivalent to

$$(E - v)^2 < E^2$$

which we can simplify to

$$v(v - 2E) < 0$$

Since v is positive, $v - 2E$ must be negative, which implies (4.6). We can now write the following function to draw arc PQ:

```
void arc8(int r)
{  int x=0, y=r, u=1, v=2*r-1, E=0;
   while (x <= y)
   {  putpix(x, y);
      x++; E += u; u += 2;
      if (v < 2 * E) {y--; E -= v; v -= 2;}
   }
}
```

Note that the statements

```
x++; E += u; u += 2;
```

applying to x are similar to the following statements, which apply to y and are executed only if (4.6) is satisfied:

```
y--; E -= v; v -= 2;
```

Function **arc8** is the basis for our final function, **draw_circle**, listed below. Besides drawing a full circle instead of only an arc, this function is also more general than **arc8** in that it can have any center C (with coordinates x_C and y_C):

```
void draw_circle(int xC, int yC, int r)
{  int x=0, y=r, u=1, v=2*r-1, E=0;
   putpix(xC, yC + r);  // Top
   putpix(xC, yC - r);  // Bottom
   putpix(xC + r, yC);  // Right
   putpix(xC - r, yC);  // Left
   while (x < y)
   {  x++; E += u; u += 2;
      if (v < 2 * E){y--; E -= v; v -= 2;}
      if (x <= y)
      {  putpix(xC + x, yC + y);  // Octant 2 (see Fig. 4.5 )
         putpix(xC - x, yC + y);  // Octant 3
         putpix(xC + x, yC - y);  // Octant 7
         putpix(xC - x, yC - y);  // Octant 6
         if (x < y)
         {  putpix(xC + y, yC + x);  // Octant 1
            putpix(xC - y, yC + x);  // Octant 4
            putpix(xC + y, yC - x);  // Octant 8
            putpix(xC - y, yC - x);  // Octant 5
         }
      }
   }
}
```

4.4 Polygon Filling

We will now deal with the problem of filling a given polygon with a given color in an efficient way. The vertices of the polygon are given by their pixel coordinates. Suppose that the coordinates of polygon $P_0P_1...P_{n-1}$ are stored in the integer array elements X[i], Y[i] (i = 0, 1,..., n–1). Then we will develop and use function **fill** to fill this polygon with the current foreground color (also used to draw lines) as follows:

```
fill(X, Y, n);
```

The method we will use is based on *scanlines*, that is, on horizontal lines of pixels. For example, to fill the entire screen, we have to draw all scanlines completely, which can be done as follows:

```
for (y=0; y<=Y__max; y++) horline(0, X__max, y);
```

(Recall that we have introduced the function **horline** at the end of Section 4.2.) Instead of drawing all scanlines, we will use only those scanlines which intersect the given polygon. If y_{min} and y_{max} are the smallest and the greatest y-coordinates of the polygon vertices, we will consider the scanlines $y = y_{min}$, $y = y_{min+1}$,..., $y = y_{max}$, and

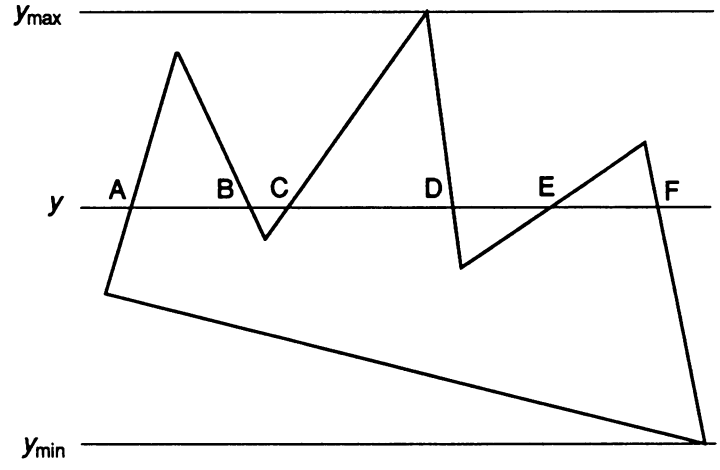


Fig. 4.7. Polygon, scanline, and points of intersection

find their points of intersection with the polygon. Figure 4.7 shows such points A, B,..., F for one particular scanline. We sort these points according to increasing x-coordinate, as shown in Fig. 4.7 from left to right. There are always an even number of points of intersection, which means that they are the endpoints of an odd number of line segments. In our example, there are five such line segments, namely AB, BC, CD, DE and EF. The first, the third, and the fifth of these lie inside the polygon and are therefore to be drawn. This is also the case in other situations: starting with the first (such as AB), we always have to draw every other line segment.

Although in principle our problem is now solved, we have yet to implement this in an efficient way, similar to the algorithm for lines, discussed in Section 4.2. Recall that x increased by 1 in each step of that algorithm, and that we repeatedly updated an 'error' D, to find out whether or not we had to alter the value of y. Starting at the lowest vertex and ending at the highest, we will be dealing with all relevant scanlines one by one, so this time it is the y-coordinate that in each step increases by 1, and the x-coordinate that may or may not change. Each of these scanlines has two or more points of intersection with the polygon sides, so we have to deal with several lines at the same time, which makes this problem more complicated than that discussed in Section 4.2. We will use linked lists for certain scanlines, as shown in Fig. 4.8.

There will be a linked list for every scanline that contains vertices from which polygon sides depart upwards. For example, there are two such sides for vertex (0, 30), at the bottom left in Fig. 4.8. For each of these, there is an element in the linked list associated with scanline $y = 30$. In general, each element in a linked list corresponds to a (non-horizontal) polygon side PQ, where we choose P and Q such that Q lies higher than P (that is, $y_Q > y_P$). There are no list elements for any horizontal sides. The start pointers of these linked lists are stored in array **table**, as shown in Fig. 4.8. The following (integer) data for line segment PQ are stored in the corresponding list element:

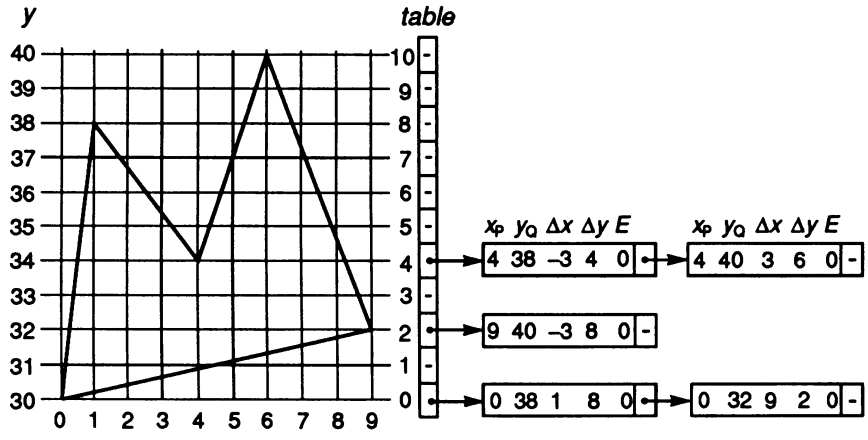


Fig. 4.8. Data structure for polygon

x_P the x-coordinate of point P, lying on the scanline in question
y_Q the y-coordinate of point Q, lying higher than P
Δx $= x_Q - x_P$
Δy $= y_Q - y_P$
E the error due to approximating a line by pixels

After building all these linked lists, we start the actual process of polygon filling. We then use another linked list, the so-called *active list*, which changes all the time as follows. We begin with an empty active list (consisting of only a sentinel) for the lowest relevant scanline ($y = y_{min}$). We now move the elements of the list starting at **table[0]** to this active list, that is, we insert these elements in the active list and delete them from the list for scanline $y = y_{min}$. In each step, we increase y by 1 and perform the following actions:

(1) We delete any elements of the active list that correspond to line segments PQ with the higher point Q on the current scanline y.

(2) We update the data members x_P and E in all remaining elements in the active list. After this, the new x_P is the best approximation of the point of intersection of scanline y and the polygon side that corresponds to the list element in question, and E is the new 'error value'.

(3) We move any elements of the linked list for scanline y (see Fig. 4.8) to the active list. Insertion is done such that the elements are in increasing order of x_P. If two elements with the same value of x_P are to be inserted, they are placed in decreasing order of their slopes. For example, point (0, 30) is the lower endpoint of sides to the points (1, 38) and (9, 32). They are to be placed in that order, because the slope of the first side is greater than that of the second.

(4) We draw horizontal segments of scanline y, the endpoints of which are given by successive pairs of x_P values, stored in the active list. Those values found

in the first and the second element of the active list apply to the first line segment. If there is a third and a fourth element, we use their x_p values to draw the second line segment, and so on.

Figure 4.9 shows three snapshots of this active list after dealing with scanlines $y = 30$, 31 and 32. Updating x_p and E, as mentioned in (2), is similar to Bresenham's algorithm as discussed in Section 4.2. Since y increases by 1, the mathematically correct value to increase x by would be the inverse of the slope, that is,

$$\frac{\Delta x}{\Delta y}$$

Instead, we try the integer value m obtained by truncating this inverted slope. The error due to using m is

$$\frac{\Delta x}{\Delta y} - m$$

We add this to the current value of the error d and again call the result d. (Initially, $d = 0$.) Increasing x once again, but this time by d, would lead to its mathematically correct value. However, d is a real number and we want to use only integers. If d is greater than 0.5, we had better increase x by 1, provided that we decrease d by 1 at the same time.

As in Section 4.2, we can switch to integers by using all denominators as multiplication factors in a consistent way. This time these denominators are Δy and 2 (occurring in the constant 0.5). Instead of the real number d we therefore use the integer

$$E = 2\Delta y \cdot d$$

and we use the test $E > \Delta y$ instead of $d > 0.5$. Similarly, we increase E by $2(\Delta x - m\Delta y)$ instead of increasing d by $\Delta x/\Delta y - m$. Finally, we decrease E by $2\Delta y$ instead of decreasing d by 1.

The way we compare slopes, as mentioned in (3), may also require some explanation. Let us distinguish the two line segments in question by the digits 1 and 2. Then it will be clear from Fig. 4.8 that the slope of line segment 1 is greater than that of line segment 2 if

$$\frac{\Delta x_1}{\Delta y_1} < \frac{\Delta x_2}{\Delta y_2}$$

Since both Δy_1 and Δy_2 are positive, we can multiply this inequality by the product of these denominators, which gives

$$\Delta x_1 \Delta y_2 < \Delta x_2 \Delta y_1$$

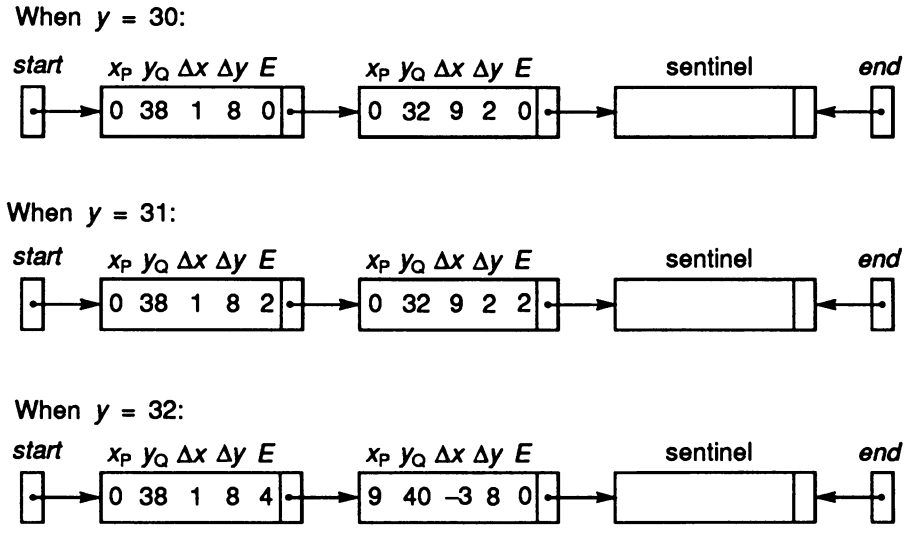


Fig. 4.9. Active list (three end situations)

This comparison has the advantage that it can be done by using only integer arithmetic. However, we have to use type **long int** in this case, because these products can be quite large. Fortunately, this comparison need not be done very often, so using type **long** will not considerably slow down our filling process.

Although function **fill** is rather long and perhaps not easy to read, it is very fast because it does not use any floating-point arithmetic. Besides filling the polygon, it also draws it. We will place this function in a module of its own; although it might seem strange to use a header file with only one function declaration, we will do this for reasons of consistency:

```
// FILL.H: Header file for polygon-fill module.
void fill(int *X, int *Y, int n);
```

```
// FILL.CPP: Polygon fill function, using integers only.

#include <stdlib.h>
#include "fill.h"
#include "grsys.h"

static void checkmem(void *p)
{  if (p == NULL) errmess("Not enough memory in 'fill'");
}
```

```
void fill(int *X, int *Y, int n)
{  int x, y, i, ymin=10000, ymax=0, j, ny, i1, xP, yP, xQ, yQ,
       temp, dx, dy, m, dyQ, E, xleft, xright;
   typedef struct element
   {  int xP, yQ, dx, dy, E; element *next;
   }  *eltptr;
   eltptr *table, p, start, end, p0, q;
   x = X[n-1]; y = Y[n-1];
   for (i=0; i<n; i++)
   {  draw_line(x, y, X[i], Y[i]);
      x = X[i]; y = Y[i];
      if (y < ymin) ymin = y;
      if (y > ymax) ymax = y;
   }
   ny = ymax - ymin + 1;
   table = new eltptr[ny]; checkmem(table);
   for (j=0; j<ny; j++) table[j] = NULL;
   for (i=0; i<n; i++)
   {  i1 = i + 1;
      if (i1 == n) i1 = 0; // i1 is i's successor
      xP = X[i]; yP = Y[i];
      xQ = X[i1]; yQ = Y[i1];
      if (yP == yQ) continue;
      if (yQ < yP)
      {  temp = xP; xP = xQ; xQ = temp;
         temp = yP; yP = yQ; yQ = temp;
      }
      p = new element; checkmem(p);
      p->xP = xP; p->dx = xQ - xP;
      p->yQ = yQ; p->dy = yQ - yP;
      j = yP - ymin;
      p->next = table[j]; table[j] = p;
   }
   start = end = new element; // Sentinel
   checkmem(start);
   for (j=0; j<ny; j++)
   {  y = ymin+j; // Build or update active edge list:
      p = start;
      while (p != end)
      {  if (p->yQ == y)
         {  // Delete list element *p:
            if ((q = p->next) == end) end = p; else *p = *q;
            delete q;
         }  else // Update list element *p:
         {  if ((dx = p->dx) != 0)
            {  x = p->xP;
```

```
                dy = p->dy;
                E = p->E;
                m = dx/dy; // Integer division!
                dyQ = 2 * dy;
                x += m; E += 2 * dx - m * dyQ;
                if (E > dy || E < -dy)
                {  if (dx > 0) {x++; E -= dyQ;}
                          else {x--; E += dyQ;}
                }
                p->xP = x;
                p->E = E;
             }
             p = p->next;
          }
       }
       // End of updating the elements (if any) of active list
       // Edges may now be added to active edge list:
       p = table[j];
       while (p != NULL)
       {  x = end->xP = p->xP; yQ = p->yQ;
          dx = p->dx; dy = p->dy; q = start;
          while (q->xP < x ||
                 q->xP == x && q != end &&
                 (long)q->dx * dy < (long)dx * q->dy)
                    q = q->next;
          p0 = p; p = p->next;
          if (q == end) end = p0; else *p0 = *q;
          q->xP = x; q->yQ = yQ;
          q->dx = dx; q->dy = dy;
          q->E = 0; q->next = p0;
       }
       // Draw line segments:
       for (p=start; p!=end; p=p->next)
       {   xleft = p->xP + 1; p = p->next;
           xright = p->xP - 1;
           if (xleft <= xright)
           {  horline(xleft, xright, y);
           }
       }
    }
    p = start;
    while (p != end)
    {  p0 = p; p = p->next; delete p0;
    }
    delete start; delete[] table;
}
```

If you want to use a general program to demonstrate this **fill** function, you could use the following one:

```
/* FILLPROG: Demonstration program for polygon filling.
   The name of an input file must be given as a program argument.
   This file must contain n pairs of pixel coordinates.
   To be linked with the modules GRSYS and FILL.
*/
#include <fstream.h>
#include <stdlib.h>
#include "grsys.h"
#include "fill.h"

int main(int argc, char *argv[])
{  const int N=400;
   int X[N], Y[N];
   int npts=0, x, y;
   if (argc < 2)
      errmess("Enter FILLPROG followed by a file name");
   ifstream ff(argv[1]);
   if (!ff) errmess("Cannot open file for input");
   for ( ; ; )
   {  ff >> x >> y;
      if (ff.eof() || ff.fail()) break;
      if (npts == N) errmess("Too many vertices");
      X[npts] = x; Y[npts++] = y;
   }
   initgr();
   fill(X, Y, npts); // This draws and fills the polygon.
   endgr();
   return 0;
}
```

Figure 4.10 shows the pixels that are written as a result of polygon filling if you start this program by entering

```
fillprog polygon.txt
```

where the file POLYGON.TXT has the following contents:

```
0 30
9 32
6 40
4 34
1 38
```

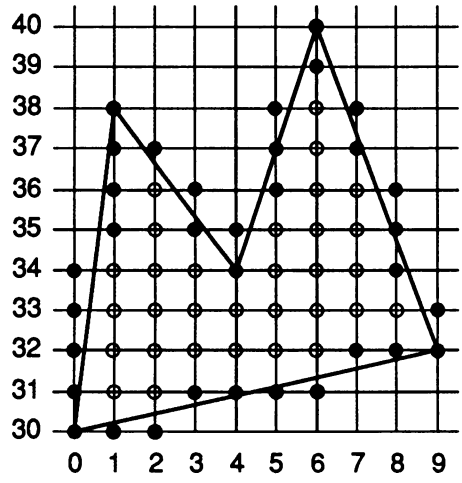


Fig. 4.10. Result of polygon filling

The open circles are pixels that are written by *filling* the polygon, while most black ones are only the result of *drawing* it. The only exceptions are the three pixels (3, 31), (7, 32) and (8, 32), which are written both by drawing and by filling. This is because the slope of the line through (0, 30) and (9, 32) is less than 1. As we discussed in Section 4.2, this line is drawn by using x as the independent variable (with step size 1). Our **fill** function, on the other hand, always uses y as the independent variable, regardless of the slope. This is why, besides filling, we also *draw* the polygon.

It goes without saying that the approximation of a polygon by pixels in Fig. 4.10 is rather poor because this polygon happens to be very small compared with the pixel size.

Exercises

4.1 Use polygon filling to draw lines that have a given thickness.

4.2 Write a function **draw_ellipse**, similar to **draw_circle**, discussed in Section 4.3, but with two parameters *rhor* and *rvert* instead of a radius *r*. This function is to draw an ellipse with horizontal and vertical axes of symmetry. The lengths of these axes are $2rhor$ and $2rvert$, respectively.

4.3 Draw Pythagoras' tree (see Section 3.7) and fill its squares and triangles with different colors, chosen at random.

4.4 The same as Exercise 4.3, except for the way squares and triangles are filled with colors. Let us assume that the largest square has sides of length a, so its area is a^2. Then there are two squares with area $a^2/2$, four with area $a^2/4$, and so on. We will say that squares with area $a^2/2^i$ are *at level i*. Write a program that produces a tree similar to the one in Exercise 4.3, but in which the color for each square is derived from its level, so that all squares at the same level have the same color.

If the tree is built in the same way as in Section 3.7, some colors will overwrite others, which makes the tree asymmetric with respect to its colors. This asymmetric appearance, illustrated by Plate 1, is due to the principle of *depth-first traversal* that is used: when a left and a right subtree are to be drawn, the left one is completed, including its subtrees, before we start with the right one. As Plate 2 shows, the appearance of the tree will be better if we use *breadth-first* traversal and eliminate recursion. In this way all squares at level i are filled before those at level $i + 1$. This can be done by means of a *queue*, in which we store data for deferred calls to function **Pythagoras** instead of performing these calls immediately. In the **main** function, we begin by placing the data (that is, the arguments) for the initial call in the queue. Then, as long as the queue is not empty, we repeatedly delete the front of the queue, using the data found there for a call to **Pythagoras**. This call will add new elements to the rear of the queue (except if the highest level is reached), and so on.

5

Perspective

5.1 Introduction

In Fig. 5.1 a two-dimensional representation of a cube is shown along with some auxiliary lines. Lines such as AB and AD in this picture are not parallel to the lower and upper edges of the paper, so one could argue that these lines are not horizontal. However, they denote horizontal edges of the cube ABCDEFGH in three-dimensional space and we will therefore briefly call them horizontal lines. For the same reason we will say that two lines such as AB and DC are parallel, implicitly thinking in three dimensions. In this terminology, parallel horizontal lines meet in a so-called *vanishing point*. All these vanishing points lie on the same line, which is called the *horizon*. Note that the horizon and vanishing points refer to the two-dimensional image space, not to the three-dimensional object space. For many centuries these concepts have been used by artists to draw realistic images of three-dimensional objects. This way of representing three-dimensional objects is usually referred to as *perspective*.

The invention of photography offered a new (and easier) way of producing images in perspective. There is a strong analogy between a camera used in photography and the human eye. Our eye is a very sophisticated instrument of which a camera is an imitation. In the following discussion the word *eye* may be replaced with *camera* if we want to emphasize that a two-dimensional hard copy is desired.

It is obvious that the image will depend on the position of the eye. An important aspect is the distance between the eye and the object, since the effect of perspective will be inversely proportional to this distance. If the eye is close to the object, the effect of perspective is strong, as shown in Fig. 5.2(a). Here we can very clearly see that in the image the extensions of parallel line segments meet.

Besides the classical and the photographic method, there is a way of producing perspective images which is based on analytical geometry. We are by now very

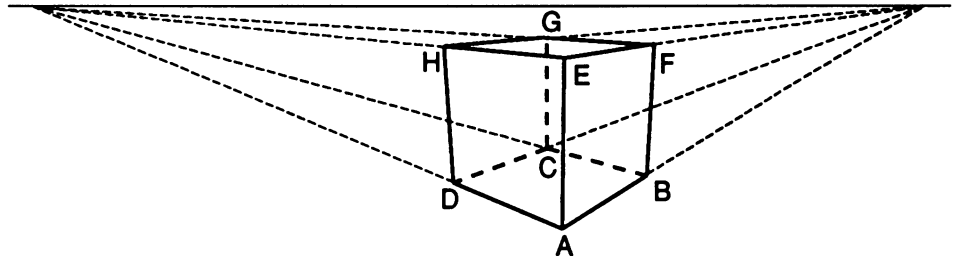
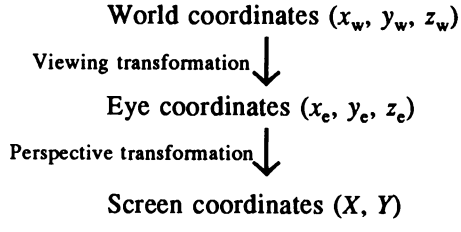


Fig. 5.1. Vanishing points on the horizon

familiar with representing points in 2-space and 3-space by their coordinates (X, Y) and (x, y, z), respectively. (We write 'n-space' for 'n-dimensional space'.)

If we want to produce a drawing in perspective, we are given a great many points $P(x, y, z)$ of the object and we want their images $P'(X, Y)$ in the picture. Thus all we need is a mapping from the *world coordinates* (x, y, z) of a point P to the *screen coordinates* (X, Y) of its central projection P'. We imagine a screen between the object and the eye E. For every point P of the object the line PE intersects the screen at point P'. It is convenient to perform this mapping in two stages. The first is called a *viewing transformation*; point P is left at its place, but we change from world coordinates to so-called *eye coordinates*. The second stage is called a *perspective transformation*. This is a proper transformation from P to P', combined with a transition from the three-dimensional eye coordinates to the two-dimensional screen coordinates:



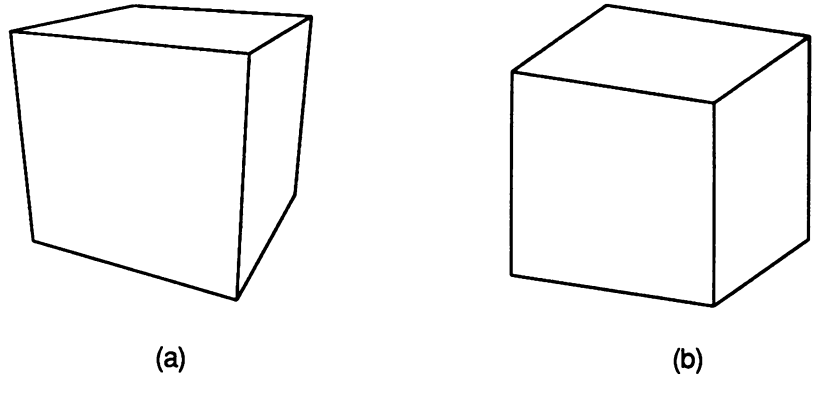


Fig. 5.2. (a) *Eye nearby*; (b) *eye far away*

5.2 The Viewing Transformation

To perform the viewing transformation we must be given not only an object but also a viewpoint E. Let us require that the world-coordinate system be right-handed. It is convenient if its origin O lies more or less centrally in the object; we then view the object from E to O. We will assume that this is the case; in practice this might require a coordinate transformation consisting of decreasing the original world coordinates by the coordinates of the central object point. We will include this very simple coordinate transformation in our program, without writing it down in mathematical notation.

Let the viewpoint E be given by its spherical coordinates ρ, θ, φ, relative to the world-coordinate system. Thus its world-coordinates are

$$\begin{aligned} x_E &= \rho \sin\varphi \cos\theta \\ y_E &= \rho \sin\varphi \sin\theta \\ z_E &= \rho \cos\varphi \end{aligned} \tag{5.1}$$

as shown in Fig. 5.3.

The direction of vector **EO** (= **–OE**) is said to be the viewing direction. From our eye at E we can only see points within some cone whose axis is EO and whose apex is E. If rectangular coordinates x_E, y_E, z_E were given, we could derive the spherical coordinates from them in the way discussed in Section 3.6.

Our final objective will be to compute the screen coordinates X, Y, where we have an X-axis and a Y-axis, lying in a screen between E and O and perpendicular to the viewing direction **EO**. This is why the eye-coordinate system, which we will deal with first, will have its x_e-axis and y_e-axis perpendicular to **EO**, leaving the z_e-axis in the direction of EO. The origin of the eye-coordinate system is viewpoint E, as shown in Fig. 5.4. Viewing from E to O, we find the positive x_e-axis pointing to the right and the positive y_e-axis upwards. These directions will enable us later to establish screen

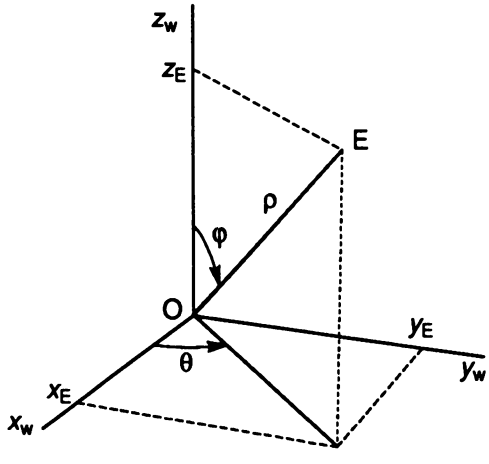


Fig. 5.3. Spherical coordinates of viewpoint E

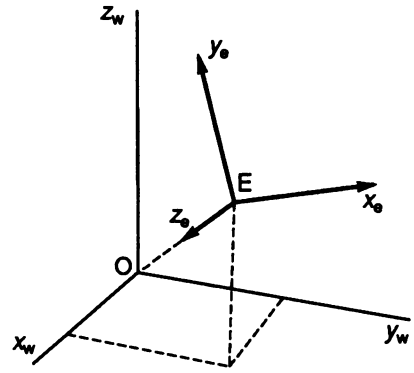


Fig. 5.4. Eye-coordinate system

axes in the same directions. The z_e-axis is chosen such that points with large positive z_e-coordinates are far away. All this is logical and convenient, but its consequence is that the eye-coordinate system is left-handed. Although this might seem odd, it is not unusual in computer graphics and it will cause no problems at all. (Note that our world-coordinate system will always be right-handed.)

The viewing transformation can be written as a matrix multiplication, for which we need the 4 × 4 *viewing matrix* V:

$$[x_e \quad y_e \quad z_e \quad 1] = [x_w \quad y_w \quad z_w \quad 1]\, V \tag{5.2}$$

To find V, we imagine this transformation to be composed of four elementary ones, for which the matrices can easily be written down. Matrix V will be the product of these four matrices. Each of the four transformations is in fact a change of coordinates and has therefore a matrix which is the inverse of the matrix that a similar point transformation would have.

(1) Moving the origin from O to E.

We perform a translation of the coordinate system such that viewpoint E becomes the new origin. The matrix for this change of coordinates is

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_E & -y_E & -z_E & 1 \end{bmatrix} \tag{5.3}$$

(Do not confuse x_E, y_E, z_E, which are the world coordinates of viewpoint E, with x_e, y_e, z_e, the eye coordinates of any point.) The new coordinate system is shown in Fig. 5.5.

(2) Rotating the coordinate system about the *z*-axis

Referring to Fig. 5.5 we now rotate the coordinate system about the *z*-axis through the angle $\frac{1}{2}\pi - \theta$ in the negative sense. This has the effect that the *y*-axis obtains the direction of the horizontal component of **OE** and that the *x*-axis becomes perpendicular to **OE**. The matrix for this change of coordinates is the same as that for a rotation of points through the same angle in the positive sense. The 3 × 3 matrix for this rotation is:

$$R_z = \begin{bmatrix} \cos(0.5\pi-\theta) & \sin(0.5\pi-\theta) & 0 \\ -\sin(0.5\pi-\theta) & \cos(0.5\pi-\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \sin\theta & \cos\theta & 0 \\ -\cos\theta & \sin\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.4}$$

The new position of the axes is shown in Fig. 5.6, where, in contrast to Fig. 5.5, the *x*- and *y*-axes are *not* parallel to the x_w- and y_w-axes. The angle $\frac{1}{2}\pi - \varphi$, indicated in Fig. 5.6, is measured in the plane through the *z*- and the z_w-axes.

(3) Rotating the coordinate system about the *x*-axis

Since the *z*-axis is to have the direction **EO**, we now rotate the coordinate system about the *x*-axis through the angle $\pi - \varphi$ ($= \frac{1}{2}\pi + (\frac{1}{2}\pi - \varphi)$, see Fig. 5.6). This

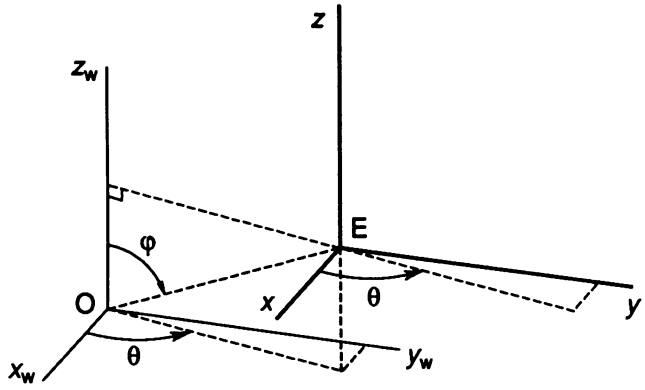


Fig. 5.5. Situation before rotation about the z-axis

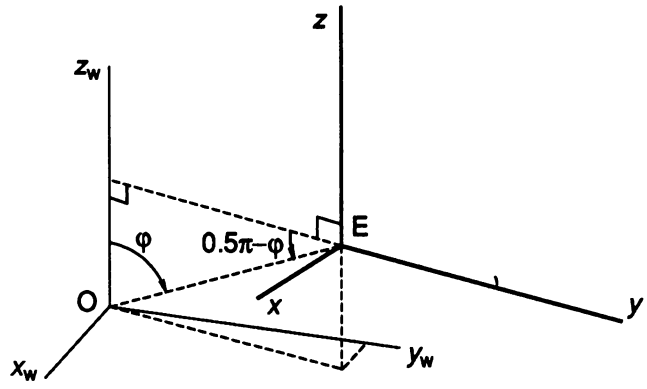


Fig. 5.6. Situation before rotation about x-axis

corresponds to a rotation of points through the angle $-(\pi - \varphi) = \varphi - \pi$. It follows from Section 3.6, Eq. (3.7), that this rotation is described by the following matrix:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\varphi-\pi) & \sin(\varphi-\pi) \\ 0 & -\sin(\varphi-\pi) & \cos(\varphi-\pi) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & -\cos\varphi \end{bmatrix} \tag{5.5}$$

The new axes are shown in Fig. 5.7. Note that the x-axis has the same direction as in Fig. 5.6, that is, it points to the reader.

(4) Changing the direction of the x-axis

In Fig. 5.7 the y-axis and the z-axis have the right positions, but the x-axis is to point in the opposite direction. Thus we need the matrix for $x' = -x$, which is:

$$M_{yz} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.6}$$

After this final transformation we have obtained the eye-coordinate system, already shown in Fig. 5.4.

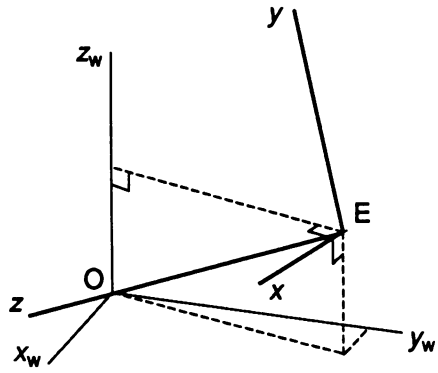


Fig. 5.7. Situation after rotating about x-axis

We can now compute the viewing matrix V as the matrix product

$$V = TR_z{}^*R_x{}^*M_{yz}{}^* \tag{5.7}$$

where the notation R^* is used for the 4×4 matrix obtained by adding a fourth row and a fourth column, containing the numbers 0, 0, 0 and 1, to the 3×3 matrix R, in the usual way. Matrix multiplication is not commutative (in general $AB \neq BA$), but it is associative, so we can write Eq. (5.7) as

$$V = T(R_zR_xM_{yz})^*$$

In this way we can deal with 3×3 matrices as long as possible. Our rather elaborate multiplication task is further relieved by introducing the abbreviations

$$\begin{aligned} a &= \cos\varphi \\ b &= \sin\varphi \\ c &= \cos\theta \\ d &= \sin\theta \end{aligned} \tag{5.8}$$

Thus $a^2 + b^2 = 1$ and $c^2 + d^2 = 1$. Using Eqs. (5.1), we rewrite Eq. (5.3) as

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\rho bc & -\rho bd & -\rho a & 1 \end{bmatrix}$$

and Eqs. (5.4) and (5.5) as

$$R_z = \begin{bmatrix} d & c & 0 \\ -c & d & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -a & -b \\ 0 & b & -a \end{bmatrix}$$

We find the following product of these matrices:

$$R_z R_x = \begin{bmatrix} d & -ac & -bc \\ -c & -ad & -bd \\ 0 & b & -a \end{bmatrix}$$

Post-multiplying this matrix by M_{yz} of Eq. (5.6) we obtain

$$R_z R_x M_{yz} = \begin{bmatrix} -d & -ac & -bc \\ c & -ad & -bd \\ 0 & b & -a \end{bmatrix}$$

We then find the desired viewing matrix V as the product of matrix T, given above, and

$$(R_z R_x M_{yz})^* = \begin{bmatrix} -d & -ac & -bc & 0 \\ c & -ad & -bd & 0 \\ 0 & b & -a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Hence

$$V = \begin{bmatrix} -d & -ac & -bc & 0 \\ c & -ad & -bd & 0 \\ 0 & b & -a & 0 \\ v_{41} & v_{42} & v_{43} & 1 \end{bmatrix}$$

where

$$v_{41} = \rho bcd - \rho bcd = 0$$

$$\begin{aligned} v_{42} &= \rho abc^2 + \rho abd^2 - \rho ab \\ &= \rho\{ab(c^2 + d^2) - ab\} \\ &= \rho(ab - ab) \\ &= 0 \end{aligned}$$

$$\begin{aligned} v_{43} &= \rho b^2c^2 + \rho b^2d^2 + \rho a^2 \\ &= \rho\{b^2(c^2 + d^2) + a^2\} \\ &= \rho\{b^2 + a^2\} \\ &= \rho \end{aligned}$$

Thus we have found

$$V = \begin{bmatrix} -\sin\theta & -\cos\varphi\cos\theta & -\sin\varphi\cos\theta & 0 \\ \cos\theta & -\cos\varphi\sin\theta & -\sin\varphi\sin\theta & 0 \\ 0 & \sin\varphi & -\cos\varphi & 0 \\ 0 & 0 & \rho & 1 \end{bmatrix} \tag{5.9}$$

We have now derived an important result. If we are given the spherical coordinates ρ, θ, φ of viewpoint E, we can compute the eye coordinates of a point from its world coordinates, using only Eqs. (5.2) and (5.9).

The viewing transformation, which we have now dealt with, is to be followed by the perspective transformation to be discussed in the next Section. However, we could also use the eye coordinates x_e and y_e, simply ignoring z_e. In that case we have a so-called *orthographic projection*. Every point P of the object is then projected into a point P′ by drawing a line from P, perpendicular to the plane through the x-axis and the y-axis. It can also be regarded as the perspective image we obtain if the viewpoint is infinitely far away. An example of such a picture is the cube in Fig. 5.2(b). Parallel lines remain parallel in pictures obtained by orthographic projection. Such pictures are very often used in practice because with conventional methods they are easier to draw than real perspective images.

On the other hand, bringing some perspective into the picture will make it much more realistic. Our viewing transformation will therefore be followed by the perspective transformation, which will involve surprisingly little computation.

5.3 The Perspective Transformation

You might have the impression that we are only half-way, and that in this section we will need as much mathematics as Section 5.2. However, most of the work has already been done. Since we will not use world coordinates in this section, there will be no confusion if we denote eye coordinates simply by (x, y, z) instead of (x_e, y_e, z_e).

In Fig. 5.8 we have chosen a point Q, whose eye coordinates are $(0, 0, d)$ for some positive number d.

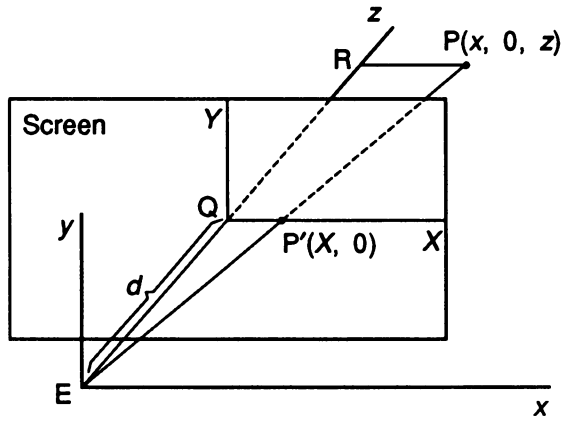


Fig. 5.8. Screen and eye coordinates

Our screen will be the plane $z = d$, that is, the plane through Q and perpendicular to the z-axis. Then the screen-coordinate system has Q as its origin, and its X- and Y-axes are parallel to the x- and y-axes. For every object point P, the image point P′ is the intersection of line PE and the screen. To keep Fig. 5.8 simple, we consider a point P whose y-coordinate is zero. However, the following equations to compute its screen coordinate X are also valid for other y-coordinates. In Fig. 5.8 the triangles EPR and EP′Q′ are similar. Hence

$$\frac{P'Q}{EQ} = \frac{PR}{ER}$$

so we have

$$\frac{X}{d} = \frac{x}{z}$$

In other words,

$$X = d \cdot \frac{x}{z} \tag{5.10}$$

In the same way we can derive

$$Y = d \cdot \frac{y}{z} \tag{5.11}$$

At the beginning of Section 5.2 we have chosen the origin O of the world-coordinate system to be a central point of the object. The origin Q of the screen-coordinate system will be central in the image because the z-axis of the eye-coordinate system is line EO, which intersects the screen at Q. Since we are using a coordinate system

with the origin in the bottom-left corner and with point (*x_center*, *y_center*) in the center of the screen, we have to replace Eqs. (5.10) and (5.11) with

$$X = d \cdot \frac{x}{z} + x\_center \tag{5.12}$$

$$Y = d \cdot \frac{y}{z} + y\_center \tag{5.13}$$

We still have to specify the distance d between viewpoint E and the screen. Roughly speaking, we have

$$\frac{d}{\rho} = \frac{image\ size}{object\ size}$$

which follows from the similarity of the triangles $EP_1'P_2'$ and EP_1P_2 in Fig. 5.9. Thus we have

$$d = \rho \cdot \frac{image\ size}{object\ size} \tag{5.14}$$

This equation should be applied to both the horizontal and the vertical directions. It should be interpreted only as a means to obtain an indication about an appropriate value for d, for the three-dimensional object may have a complicated shape and it may not be clear how its size is to be measured. We then use a rough estimation of the object size, such as the maximum of its length, width and height. The image size in Eq. (5.14) should be taken somewhat smaller than the screen. A more sophisticated way of obtaining a desired image size will be discussed in Section 5.5.

The remaining part of this section is devoted to obtaining a better insight into concepts such as vanishing points and horizon. In Fig. 5.10 we have a viewpoint E and a screen ABCD.

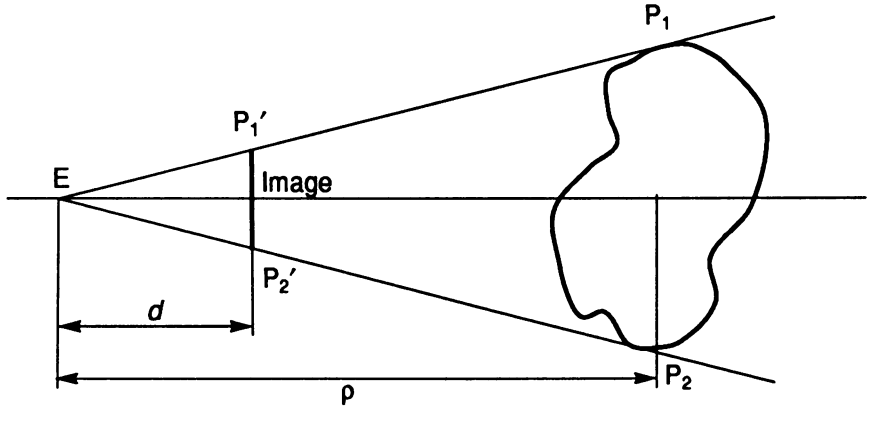


Fig. 5.9. Image size and object size

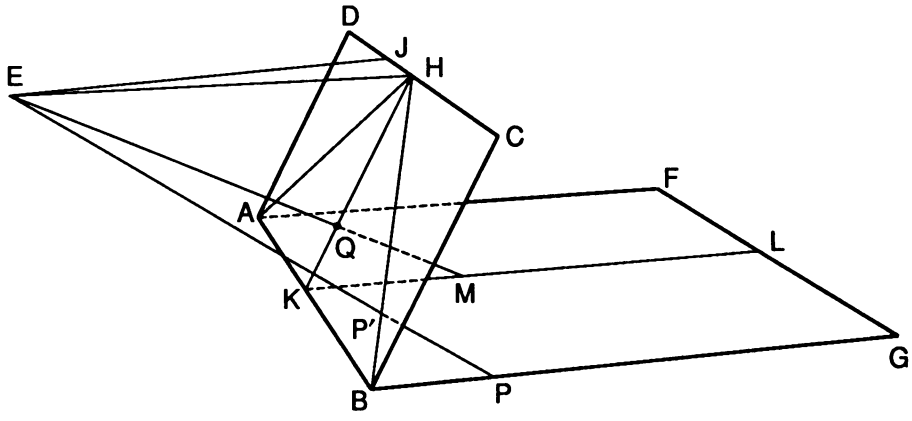


Fig. 5.10. Vanishing point H

The viewing direction is again from E to Q. Lines AF, BG, EH are parallel, and we consider them to be horizontal. We imagine a plane through the parallel lines EH and BG. This plane and the screen have BH as their line of intersection. Thus every point P on BG has its central projection P′ on BH, E being the center of projection. If we let P move away from B, towards infinity, its projection P′ will approach H. This means that H is the vanishing point of the line through B and G. In terms of projective geometry, H is the projection of the 'infinite point' of BG. In geometry parallel lines are said to intersect at an *infinite point*. The *vanishing point* of a line is the projection (that is, the image) of the infinite point of that line. The *horizon* is the set of vanishing points of all possible horizontal lines. In Fig. 5.10 the parallel lines BG and AF have the same infinite point, so H is also the projection of the infinite point of AF, and line CD is the horizon. If we take a line with a different direction, but also horizontal, such as line BF, this also has its vanishing point on line CD. It is found as the intersection point of line CD and a line through E parallel to the given horizontal line. Every point J on the horizon CD is the vanishing point of all lines that are parallel to EJ.

Not only horizontal lines have vanishing points. In Fig. 5.11, we have the vertical lines CA and DB. Point F is their vanishing point. It is the point of intersection of the vertical line through E and the screen. Line segments CA and DB have projections CA′ and DB′, which are not parallel.

We do not always appreciate a strong perspective effect for vertical lines but we often prefer pictures where vertical lines are represented by almost vertical lines. This is so because we are accustomed to horizontal or nearly horizontal viewing directions. Some people even get dizzy if they look a long way vertically! Artists sometimes produce 'pseudo-perspective' pictures where vertical lines are represented by exactly vertical lines, even if the viewing direction is not horizontal. In the latter case, such a picture differs from what we actually see but, curiously enough, it seems to be all right. An example is Fig. 5.12(b). In Section 5.6 we will revisit this phenomenon, and show that we can also produce such pseudo-perspective pictures (see also Fig. 5.21).

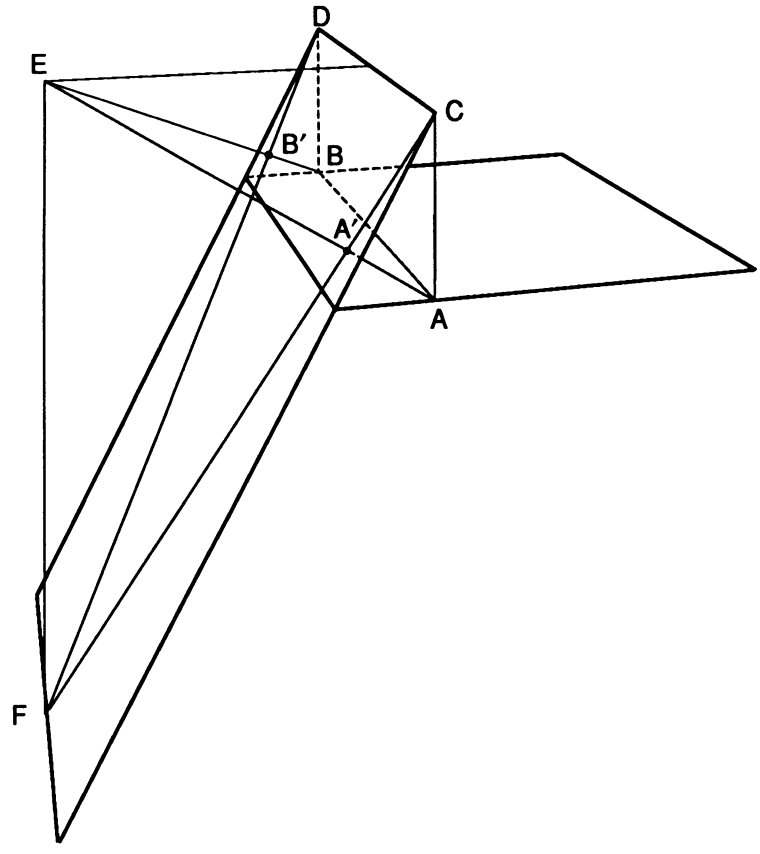


Fig. 5.11. Vanishing point F of vertical lines

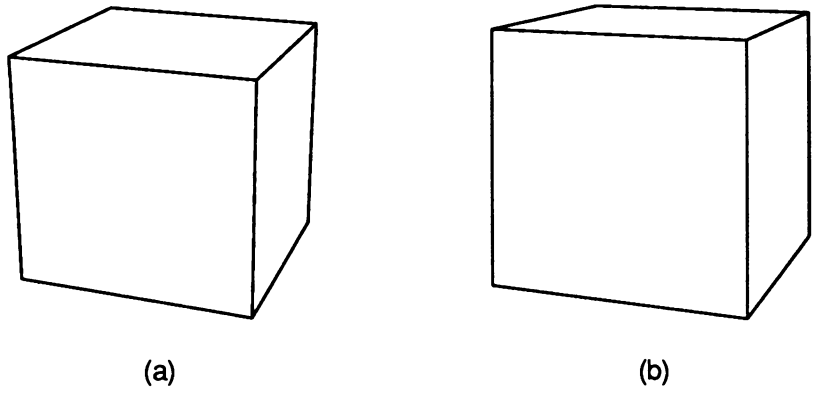


Fig. 5.12. (a) *Perspective*; (b) *pseudo-perspective*

Fig. 5.13. Ten cubes parallel to the screen

In general, however, it is a good idea to choose the viewpoint not too close to the object, especially not if the angle φ, indicated in Fig. 5.3, differs much from 90°. This is a practical way of avoiding too strong a perspective effect for vertical lines.

Another somewhat controversial subject is the representation of lines that are parallel to the screen. They will be represented by parallel lines in the picture. For example, consider Fig. 5.13, where we have ten cubes that are horizontally in line. The viewing direction is horizontal, so $\varphi = 90°$.

It seems that the cubes on the extreme left and right are larger than those in the middle. Actually, the front faces of all ten cubes are squares with the same size, not only in reality but also in this picture. However, the cubes in the middle are nearer than the others, so we may argue that their images should be larger. But this picture is obtained by rigorous geometric rules deserving our confidence. The paradox is resolved by observing that Fig. 5.13 is unrealistic with respect to the way we view objects. The construction of our eye is such that we only see points within a certain cone, whose axis is the viewing direction EO. An important aspect of this cone is the angle α indicated in Fig. 5.14.

Eyes and cameras allow only α-values that do not exceed a certain α_{max}, say, 45°. However, we have not limited the actual angle α, indicated in Fig. 5.14 and roughly satisfying

$$\tan \alpha = \frac{0.5 \times \textit{object size}}{\rho}$$

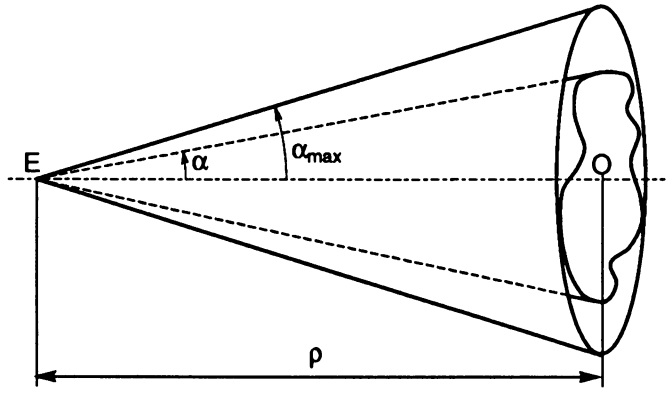


Fig. 5.14. Viewing cone

so again the cause of the trouble is a viewing distance ρ that is too small. If we choose ρ large enough, the angle α will be sufficiently small to avoid these problems. If we are not yet satisfied, there is another solution, namely, to replace a flat screen with part of a sphere whose center is E. In this way we can allow large angles α, but then we no longer have flat pictures. These spherical pictures themselves can be projected onto a plane. Readers who are interested in such unorthodox projection methods are referred to the work of the great graphical artist M.C. Escher (1972). We will restrict ourselves to flat pictures and to rather small values of α.

A program module for perspective

It will be convenient to deal with a new type for points and vectors in 3-space. As usual in C++ program development, we declare this new type, **vec3**, in a header file, together with some useful operators for three-dimensional vectors and two functions related to perspective projection. This header file is D3.H, listed below:

```
// D3.H: Header file for 3D vector operations and
//       perspective functions

#include "fstream.h"
struct vec3
{  float x, y, z;
   vec3(double xx, double yy, double zz){x = xx; y = yy; z = zz;}
   vec3(){x=0; y=0; z=0;}
};

istream &operator>>(istream &ff, vec3 &P);

vec3 operator+(vec3 &u, vec3 &v);
vec3 operator-(vec3 &u, vec3 &v);
vec3 operator*(double c, vec3 &v);

vec3 &operator+=(vec3 &u, vec3 &v);
vec3 &operator-=(vec3 &u, vec3 &v);
vec3 &operator*=(vec3 &v, double c);

void coeff(double rho, double theta, double phi);
void eyecoord(vec3 &pw, vec3 &pe);
void perspective(vec3 &p, float *px, float *py);

double dotproduct(vec3 &a, vec3 &b);
double abs(vec3 &v);
vec3 crossproduct(vec3 &a, vec3 &b);
```

We will see how to use this header file shortly in a program to draw a cube. Since this file is very general, it is listed here, rather than in the next section, which is about drawing cubes. This also applies to the associated implementation file, D3.CPP, which is to be compiled separately; the resulting object file can be linked together with application modules, such as those to be discussed shortly:

```
/* D3.CPP: Implementation file for 3D vector operations
           and perspective functions
*/
#include <math.h>
#include <fstream.h>
#include "d3.h"

istream &operator>>(istream &ff, vec3 &P)
{  return ff >> P.x >> P.y >> P.z;
}

vec3 operator+(vec3 &u, vec3 &v)
{  return vec3(u.x + v.x, u.y + v.y, u.z + v.z);
}

vec3 operator-(vec3 &u, vec3 &v)
{  return vec3(u.x - v.x, u.y - v.y, u.z - v.z);
}

vec3 operator*(double c, vec3 &v)
{  return vec3(c * v.x, c * v.y, c * v.z);
}

vec3 &operator+=(vec3 &u, vec3 &v)
{  u.x += v.x; u.y += v.y; u.z += v.z;
   return u;
}

vec3 &operator-=(vec3 &u, vec3 &v)
{  u.x -= v.x; u.y -= v.y; u.z -= v.z;
   return u;
}

vec3 &operator*=(vec3 &v, double c)
{  v.x *= c; v.y *= c; v.z *= c;
   return v;
}

static double v11, v12, v13, v21, v22, v23, v32, v33, v43;
```

```
void coeff(double rho, double theta, double phi)
{  double costh, sinth, cosph, sinph;
   // Angles in radians:
   costh = cos(theta); sinth = sin(theta);
   cosph = cos(phi); sinph = sin(phi);
   v11 = -sinth; v12 = -cosph*costh; v13 = -sinph*costh;
   v21 = costh;  v22 = -cosph*sinth; v23 = -sinph*sinth;
                 v32 = sinph;        v33 = -cosph;
                                     v43 = rho;
}

void eyecoord(vec3 &pw, vec3 &pe)
{  pe.x = v11 * pw.x + v21 * pw.y;
   pe.y = v12 * pw.x + v22 * pw.y + v32 * pw.z;
   pe.z = v13 * pw.x + v23 * pw.y + v33 * pw.z + v43;
}

void perspective(vec3 &p, float *px, float *py)
{  vec3 pe;
   eyecoord(p, pe);
   *px = pe.x/pe.z;
   *py = pe.y/pe.z;
}

double dotproduct(vec3 &a, vec3 &b)
{  return a.x * b.x + a.y * b.y + a.z * b.z;
}

double abs(vec3 &v)
{  return sqrt(v.x * v.x + v.y * v.y + v.z * v.z);
}

vec3 crossproduct(vec3 &a, vec3 &b)
{  return vec3(a.y * b.z - a.z * b.y,
               a.z * b.x - a.x * b.z,
               a.x * b.y - a.y * b.x);
}
```

Function **coeff** computes all nonzero coefficients of matrix V, derived in Section 5.2. It is to be called only once, in contrast to function **perspective**, which uses both matrix V and the perspective transformation, discussed in this section. Note that the variables v_{11}, v_{12},... are **static**; the 'user' of module D3 does not have direct access to them, and can use the same variable names for other purposes. Using simple variables for the (nonzero) elements of matrix V makes function **eyecoord** faster than working with a two-dimensional array would be.

5.4 A Program to Draw Cubes

When we are designing a program, we have to decide how general it should be. There are two extreme possibilities. On the one hand, we can write a very special program, which does not read any input data at all and can only produce one result, say, a picture. On the other hand, we can develop a very general program, which can draw any picture, provided that a file with input data is given. Between these extreme cases there are programs that produce pictures of a certain type, after having read a limited amount of input data, sometimes called parameters.

In this section we will discuss a program that can produce any perspective image of a cube in the form of a *wire frame*. As this term suggests, this means that all edges are drawn in the picture. The input data of our program consists of

(1) The following three spherical coordinates (see Fig. 5.3):
the viewing distance ρ = EO;
the angle θ, measured horizontally from the x-axis;
the angle φ, measured vertically from the z-axis.

(2) The distance d between the screen and the viewpoint. This distance is related to the size of the cube, which follows from the coordinates of its eight vertices.

The origin O of the world coordinate system is chosen at the center of the cube, as in Fig. 5.15, and the cube has edges of length 2. Thus, for example, the coordinates of point A are 1, –1 and –1.

Since we are also interested in objects other than cubes, we separate the general perspective program aspects from those which are specific to this particular cube program. For perspective programs in general we use module D3, discussed in the previous section. The part that is specific to a cube is listed below:

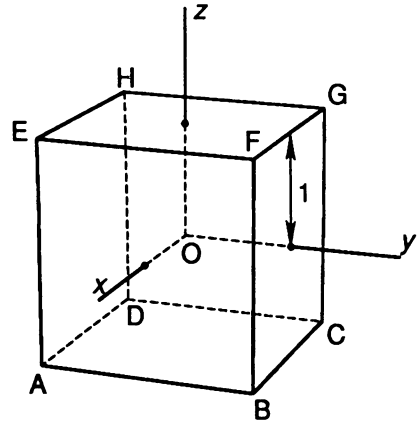


Fig. 5.15. Cube and world-coordinate system

```
// CUBE: Drawing a wire frame model of a cube in perspective
//       To be linked together with GRSYS and D3
#include <math.h>
#include <ctype.h>
#include "grsys.h"
#include "d3.h"

float d; // Screen distance

void screencoord(vec3 &p, float *px, float *py)
{  float x, y;
   perspective(p, &x, &y);
   *px = d * x + x_center;
   *py = d * y + y_center;
}

void mv(vec3 &p)
{  float x, y;
   screencoord(p, &x, &y); move(x, y);
}

void dw(vec3 &p)
{  float x, y;
   screencoord(p, &x, &y); draw(x, y);
}

int main()
{  float rho, theta, phi, pidiv180=atan(1)/45;
   char ch;
   cout << "Viewing distance rho = EO: "; cin >> rho;
   cout << "Enter two angles, in degrees.\n";
   cout << "Theta, measured horizontally from the x-axis: ";
   cin >> theta; theta *= pidiv180; // Radians
   cout << "Phi, measured vertically from the z-axis: ";
   cin >> phi; phi *= pidiv180;     // Radians
   cout << "Default screen distance? (Y/N): ";
   cin >> ch; ch = toupper(ch);
   if (ch != 'Y')
   {  cout << "Distance between screen and eye "
      "(the image size will be proportional to this value): ";
      cin >> d;
   }
   coeff(rho, theta, phi);
   initgr();
   if (ch == 'Y')
   {  d = rho * (y_max - y_min)/3.5;
```

```
        // d/rho = imagesize/objectsize
        // imagesize:  (y_max - ymin)
        // objectsize: about 3.5
    }
    vec3 A(1, -1, -1), B(1, 1, -1),
         C(-1, 1, -1), D(-1, -1, -1),
         E(1, -1, 1),  F(1, 1, 1),
         G(-1, 1, 1),  H(-1, -1, 1);
    mv(A); dw(B); dw(C); dw(G);
    dw(H); dw(E); dw(A); mv(B);
    dw(F); dw(G); mv(F); dw(E);
    mv(A); dw(D); dw(C); mv(D); dw(H);
    endgr();
    if (ch == 'Y')
    {  cout << "The default screen distance was "
            << d << endl;
    }
    return 0;
}
```

As discussed in Section 5.3, providing a suitable value for the screen distance d is not always easy. Program CUBE therefore allows us to use a default value d, computed on the basis of Eq. (5.10). If this option is chosen, this value d is displayed at the end of the program. This is useful if, after all, we want to run the program once again with a different d value. Remember, the image size will be proportional to this value d. Program CUBE has been used to produced the cubes shown in Fig. 5.16.

Note that in the second picture we view the cube from the bottom and that in the third the projections of sides and diagonals happen to lie on the same lines. These cubes can only be recognized with some difficulty. Things would have been much better if we had drawn solid objects instead of wire-frame models. This will be the subject of the remaining chapters, where we will use algorithms to find out whether a line segment or surface is visible from the viewpoint. However, if we take some care in choosing the viewpoint, we can draw not too complicated solid objects by omitting hidden lines ourselves. For the cube in Fig. 5.15 this is done by choosing the viewpoint such that only the squares ABFE, BCGF, EFGH are visible and by omitting the edges AD, CD, DH. We then simply omit the program line immediately preceding the call to function **endgr** in program CUBE. A version of CUBE modified in this way was actually used to produce Fig. 5.2(a) and (b).

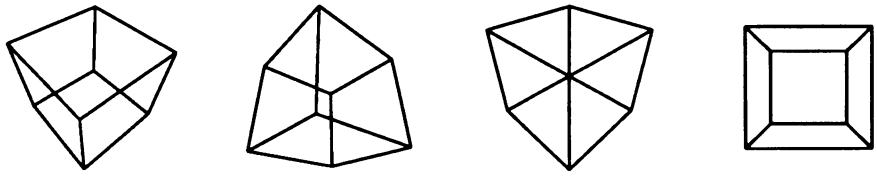


Fig. 5.16. Cubes seen from different viewpoints

5.5 Drawing Wire-frame Models

We will now consider a program which can produce a perspective picture of any wire-frame model. By omitting certain line segments, we can also produce pictures of simple solid objects, as we did for a cube in Section 5.4. As before, we will use a viewpoint E and a central object point O, such that EO is the viewing direction. It would, however, not be convenient for the user if this point O always had to coincide with the origin of his world coordinate system. We will only demand the z-axis to be vertical. The user will have to specify the coordinates x_O, y_O, z_O of central object point O, expressed in his own coordinate system. When the user's coordinates x_w, y_w, z_w are read, they are converted to 'internal world coordinates' x, y, z, with O as their origin, by means of a simple coordinate transformation:

$$x = x_w - x_O$$
$$y = y_w - y_O$$
$$z = z_w - z_O$$

Then the user has to specify the spherical coordinates ρ, θ, φ of viewpoint E relative to the new origin O, as shown in Fig. 5.3. This program will also use the module VIEWPORT, discussed in Section 2.6, which performs automatic scaling so that the image will be about as large as our screen permits. Recall that we can use the functions **initwindow**, **append** and **genplot** in that order. Function **append** computes the window boundaries and appends coordinates to a queue, which is used later by **genplot**. To specify the line segments, we imagine movements in 3-space, which are associated with corresponding pen movements when drawing the picture. For every 'pen move', a code

0 = *pen up*
1 = *pen down*

is given, as discussed in Section 2.6 for 2-space. To draw the two adjacent line segments PQ and QR, we have to specify three input lines of the following structure:

x_P	y_P	z_P	0
x_Q	y_Q	z_Q	1
x_R	y_R	z_R	1

Before presenting the program, we will consider an example of a complete set of input data for Fig. 5.10 of Section 5.3. Drawing this figure by hand would not be an easy task. For example, it would be a problem to determine the position of point M on line KL, such that in 3-space EM is perpendicular to plane BCDA. By contrast, the perspective image of Fig. 5.10 is easy to produce by means of our program GPERSF. Let us take K as the origin of the user's coordinate system. We let the positive x-axis of this system pass through B and the positive y-axis through L. Viewed from the positive x-axis, the yKz-plane is shown in Fig. 5.17.

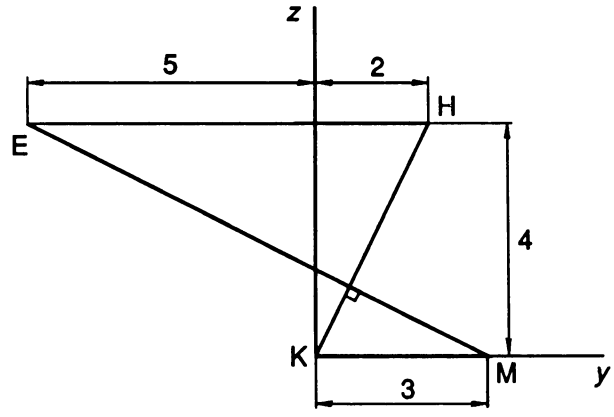


Fig. 5.17. View from positive x-axis

In 3-space, K is in the middle of segment AB, whose length is 10. We can now give the complete set of input data for the picture we wish to draw. The comments between parentheses must not really be present in the input file:

```
 5    0    0    0        (move to B)
 5    2    4    1        (draw BC)
-5    2    4    1        (draw CD)
-5    0    0    1        (draw DA)
 5    0    0    1        (draw AB)
 5    9    0    1        (draw BG)
-5    9    0    1        (draw GF)
-5    0    0    1        (draw FA)
 0   -5    4    0        (move to E)
 0    2    4    1        (draw EH)
 0    0    0    1        (draw HK)
 0    9    0    1        (draw KL)
 0   -5    4    0        (move to E)
 0    3    0    1        (draw EM)
```

The output resulting from these data is shown in Fig. 5.18. The name of the input file will be given as a program argument. Optionally, the name of an HP-GL output file can be supplied as a second program argument. For example, if the above data are located in file PLANES.DAT and we want an HP-GL output file PLANES.HPG, we can start the program as follows:

```
gpersf planes.dat planes.hpg
```

If we omit **planes.hpg** in this command line, we obtain the same result on the screen but no output file. Program GPERSF is easy to use and quite useful for producing all

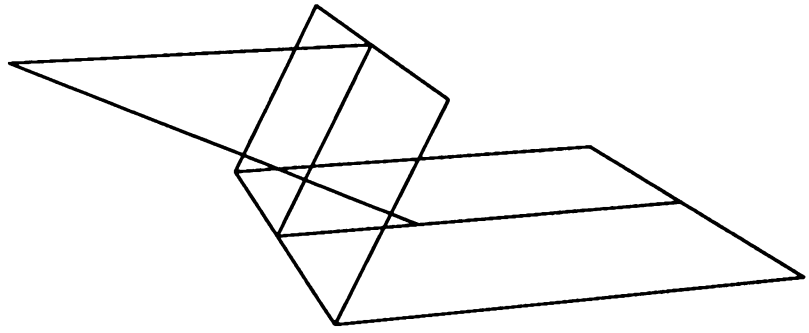


Fig. 5.18. Sample output of program GPERSF

sorts of perspective drawings. Thanks to the availability of the modules GRSYS, D3 and VIEWPORT (see Section 2.6), it is nevertheless very short:

```
/* GPERSF: General program for perspective, based
      on an input file with lines of the form
         x  y  z  drawcode
      where drawcode is 0 for 'move' and 1 for 'draw'.
      Automatic scaling will take place, so the unit
      of measurement used for x, y, z is irrelevant.
      The name of the input file is given as a program
      argument. The user may also supply the name
      of an HP-GL output file as a second program argument.
      To be linked with GRSYS, D3 and VIEWPORT.
*/
#include <fstream.h>
#include <math.h>
#include "grsys.h"
#include "d3.h"
#include "viewport.h"

int main(int argc, char *argv[])
{  float rho, theta, phi, pidiv180=atan(1)/45;
   int code;
   vec3 p, pcobj;
   float xscr, yscr;
   if (argc < 2)
      errmess("Use name of input file as a program argument");
   cout << "Enter x, y and z of central object point: ";
   cin >> pcobj;
   cout << "Enter rho, theta (both in degrees) and phi: ";
   cin >> rho >> theta >> phi;
   theta *= pidiv180; phi *= pidiv180;
```

```
    coeff(rho, theta, phi);
    ifstream ff(argv[1]);
    if (!ff) errmess("Input file not available");
    initwindow();
    for ( ; ; )
    {  ff >> p >> code;
       if (ff.eof()) break;
       perspective(p-pcobj, &xscr, &yscr);
       append(xscr, yscr, code);
       // Append this plot instruction to list
    }
    // Draw line segments according to list
    if (argc == 2) initgr(); else initgr(argv[2]);
    genplot();
    endgr();
    return 0;
}
```

5.6 Viewing Direction, Infinity, Vertical Lines

We will use program GPERSF to discuss some interesting new aspects. Suppose that an object is very large in the x-direction, such as the beam in Fig. 5.19, whose dimensions are $200 \times 2 \times 2$. This example is interesting because of the central object point O that we must specify to determine the viewing direction EO. Up to now we have chose O at the center of the object. What we actually need, however, is a point O whose image point lies at the center of the image. Often this does not make much difference, but sometimes it does. The situation is illustrated in Fig. 5.20, where in the picture point O′ lies in the middle of Q′U′, but the original point O does not lie in the

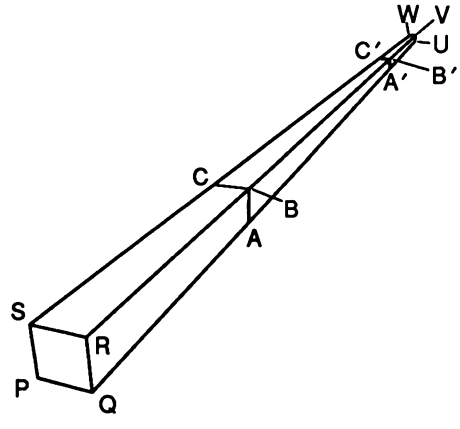


Fig. 5.19. Long beam

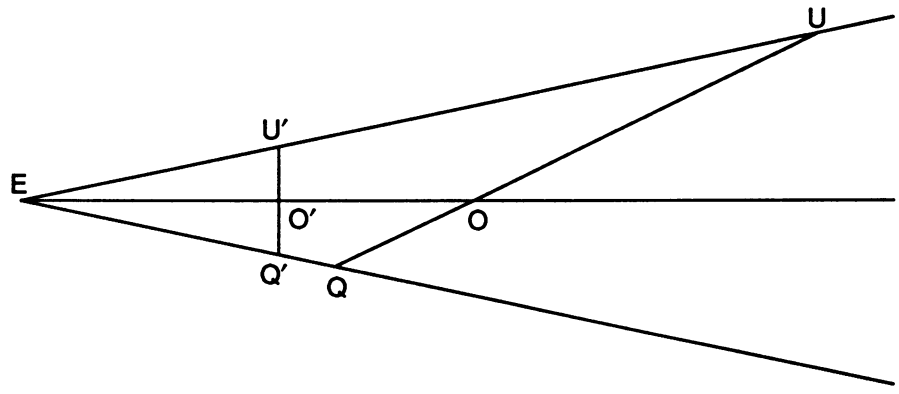


Fig. 5.20. Object point O not at center of object

middle of QU. If the middle of QU had been chosen for O, the viewing direction would have differed significantly from the current viewing direction. In cases like this we had better omit the word *central* in 'central object point', as was done in the caption of Fig. 5.20.

Returning to the beam in Fig. 5.19, it is clear that point O should not be chosen in the middle of the beam, but much nearer to the eye. This is possible because in our programs the object point is given by the user rather than computed as the object center. In the case of a landscape, finite line segments in the picture may result from infinite lines in reality. For such an infinite object it does not make sense to speak about an object center, although there is a viewing direction EO, so it does make sense to define an object point O. The beam in Fig. 5.19 is not exactly a landscape, but its length suggests infinity; if this is not clear, this length should be given a much greater value than 200. In this beam we chose O(–15, 1, 1). Except for the letters P, Q,..., this picture was actually produced by program GPERSF. The intersection of the beam with the plane $x = -15$ is denoted by ABC, and the intersection with $x = -100$ by A′B′C′. So O lies in ABC, but A′B′C′ is in the middle of the beam. Here are the contents of the input file for GPERSF, used to produce Fig. 5.19:

```
   0   2   0   0   (move to Q)
   0   2   2   1   (draw QR)
   0   0   2   1   (draw RS)
   0   0   0   1   (draw SP)
   0   2   0   1   (draw PQ)
-200   2   0   1   (draw QU)
-200   2   2   1   (draw UV)
-200   0   2   1   (draw VW)
   0   0   2   1   (draw WS)
   0   2   2   0   (move to R)
-200   2   2   1   (draw RV)
 -15   2   0   0   (move to A)
```

```
-15   2   2    1    (draw AB)
-15   0   2    1    (draw BC)
-100  2   0    0    (move to A')
-100  2   2    1    (draw A'B')
-100  0   2    1    (draw B'C')
```

The following numbers were entered on the keyboard:

```
-15    1    1        (coordinates of object point O)
 30   20   70        (ρ, θ, φ)
```

Another aspect deserving our special attention is the representation of vertical lines, briefly discussed in Section 5.3. In Fig. 5.11 we saw the projected vertical lines meet at vanishing point F. We also compared the cube representations in Figs. 5.12(a) and 5.12(b). If the center of the cube were automatically taken as object point O, our program would not be able to produce Fig. 5.12(b). However, we can place point O above the cube, such that the viewing direction EO is horizontal. The screen will then be vertical, since it is perpendicular to the viewing direction. This means that the vertical cube edges will be parallel to the screen. Their projections in the picture will therefore be exactly vertical. The curious aspect is that although the viewing direction is horizontal, we view the upper plane of the cube from above. The picture that we obtain looks quite natural, unless we exaggerate and choose the viewpoint too high. Figures 5.21(a) and (b) show examples of both cases. The lines in Figs. 5.21(a) and (b) were drawn by GPERSF, using the following integers in an input file:

```
 1   1  -1    0      (move to P)
-1   1  -1    1      (draw PQ)
-1   1   1    1      (draw QU)
 1   1   1    1      (draw UT)
 1   1  -1    1      (draw TP)
 1  -1  -1    1      (draw PS)
 1  -1   1    1      (draw SW)
 1   1   1    1      (draw WT)
 1  -1   1    0      (move to W)
-1  -1   1    1      (draw WV)
-1   1   1    1      (draw VU)
 2   0   0    0      (x-axis)
 1   0   0    1
 0   2   0    0      (y-axis)
 0   1   0    1
 0   0   2    0      (z-axis)
 0   0   1    1
```

The same spherical coordinates $\rho = 5$, $\theta = 30°$, $\varphi = 90°$ were used for Fig. 5.21(a) and (b). The only difference is the specification of the object point:

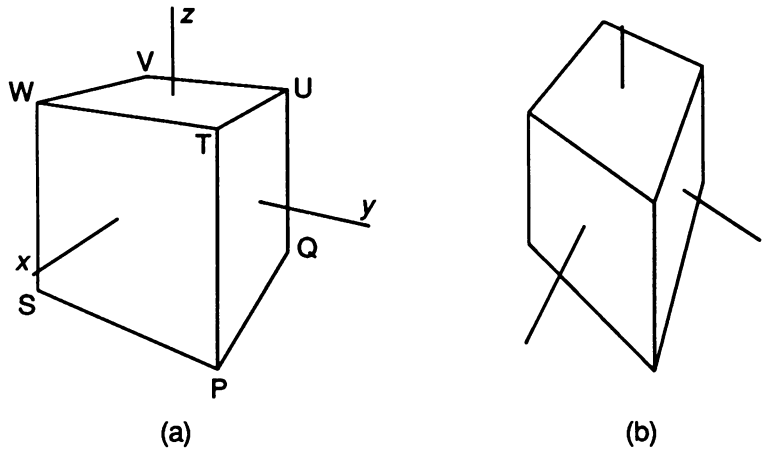


Fig. 5.21. (a) *Object point slightly above cube*; (b) *object point too high*

```
0   0   2          (object point in Fig. 5.21(a))
0   0   6          (object point in Fig. 5.21(b))
```

Figure 5.21(b) is not a realistic picture of a cube. On the other hand, Fig. 5.21(a) is quite acceptable. Some will even prefer it to the cubes in Figs. 5.2(a) and 5.12(a), in which vertical edges have a vanishing point. If a cube is drawn in perspective by hand, it is customary to leave vertical lines vertical, as in Fig. 5.21(a). The other representations are more difficult to produce manually. With the computer they are produced just as easily and the user can have it as he or she likes.

Exercises

In the following exercises, hidden edges are to be omitted in the way we discussed at the end of Section 5.4.

5.1 Write a program to draw the stairs of Fig. 5.22. The number of stairs (n) should be variable. (We have $n = 3$ in Fig. 5.22.)

5.2 Write a program to draw a pyramid consisting of cubes, as shown in Fig. 5.23. The number n specifying the height in cubes is to be read by the program. (We have $n = 3$ in Fig. 5.23.)

5.3 Write a program to draw the object of Fig. 5.24. This is a cube from which several small cubes have been removed. The dimensions of the small cubes are $1/n$ times those of the original one. The number n is variable. (We have $n = 3$ in Fig. 5.24.)

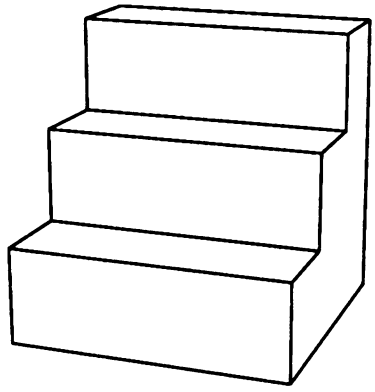


Fig. 5.22. Stairs

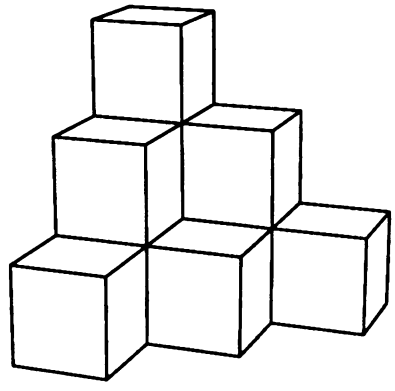


Fig. 5.23. Pyramid of cubes

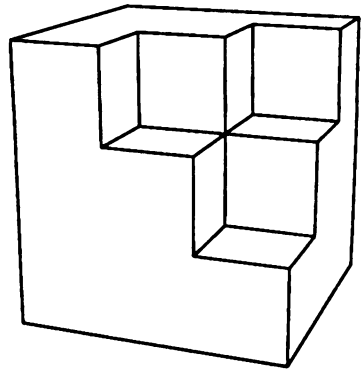


Fig. 5.24. Reduced cube

6

Hidden-line Elimination

6.1 Backfaces and Convex Polyhedra

In this chapter we will deal with *polyhedra*, that is, with solid objects, bounded by polygons. In our terminology an *object* is either one polyhedron or several of them. We will represent such objects by drawing line segments here (in contrast to Chapter 7, where we will focus on surfaces). These line segments will be drawn only as far as they are visible. This chapter explains some principles that form the basis of program module HIDELINE, which can be found in Appendix A. To obtain an executable version (with the same name), it must be linked together with the modules GRSYS (see Appendix C), D3 (see Section 5.3) and TRIANGUL (see Section 3.5). Program HIDELINE is rather complex, which is partly due to some techniques used to improve its performance. It is fast and easy to use, so most of this chapter can be skipped by those who only want to *use* this program. On the other hand, if you want to know how it works, you should study this chapter very carefully.

A polyhedron is said to be *convex* if, for any two vertices A and B that belong to it, the entire line segment AB belongs to that polyhedron. Eliminating hidden edges of a convex polyhedron can be done efficiently by using the notion of a *backface*, which is invisible because of the current viewing direction. We can detect backfaces by examining the orientation of their vertices. Each face of a polyhedron is a polygon; we will consistently specify such a polygon by giving a list of its vertex numbers, in counter-clockwise order when viewed from outside the polyhedron. Then, with a given viewpoint, such a polygon is a backface if the orientation of the corresponding list of image points is no longer counter-clockwise. We will illustrate this by means of an example, which also shows the form of the input data that we will use to describe three-dimensional objects throughout this book. The cube shown in Fig. 6.1 is a convex polyhedron. Assuming that its edges have length 10, we specify it as follows:

```
1   10    0    0
2   10   10    0
3    0   10    0
4    0    0    0
5   10    0   10
6   10   10   10
7    0   10   10
8    0    0   10
Faces:
1  2  6  5.
2  3  7  6.
3  4  8  7.
4  1  5  8.
5  6  7  8.
1  4  3  2.
```

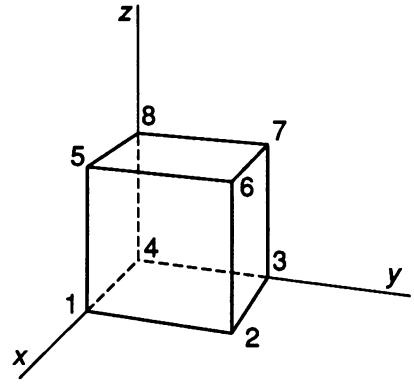


Fig. 6.1. A cube as a convex polyhedron

The above set of input data consists of two parts. First, we list all vertices by their integer vertex numbers (1, 2,...), each followed by the x-, y-, and z-coordinates of the vertex in question. Then, after the delimiter **Faces:**, we specify the vertex numbers of each face, in counter-clockwise order when viewed from the outside. Three of the six faces are visible, the other three are backfaces. For example, face 3 4 8 7 is a backface. This can be detected in our program by the fact that this list of vertices has clockwise orientation in the image, that is, in Fig. 6.1, while its orientation is counter-clockwise if we view the cube from the outside, which here means from the back.

Note that the edges of the polyhedron are not explicitly specified in the above input data but are simply given as edges of polygons. As a consequence, they occur twice. For example, edge (5, 6) occurs both in the front face (1, 2, 6, 5) and in the top face (5, 6, 7, 8). These faces are both visible, and it would not be efficient to draw their common edge (5, 6) twice. Edge (2, 3), on the other hand, is shared by the visible face (2, 3, 7, 6) and the backface (1, 4, 3, 2). Since backfaces will be ignored, no special test is required to prevent this edge from being drawn twice. Finally, (3, 4)

is an invisible edge, which will not be drawn because it belongs to two backfaces (3, 4, 8, 7) and (1, 4, 3, 2).

Besides the above input data, we also need to know the position of a central object point, denoted by O, and viewpoint E. These data can best be read from the keyboard in the form of the Cartesian coordinates x_O, y_O, z_O and the spherical coordinates ρ, θ, φ, shown in Fig. 5.3. Point O is in fact the origin of a new coordinate system with axes parallel to the user's world-coordinate axes, and the viewing direction is EO. Suitable values for Fig. 6.1 are $x_O = y_O = z_O = 5$, and $\rho = 50$, $\theta = 20°$, $\varphi = 70°$.

It is an instructive exercise to implement the ideas outlined above in a complete program (see Exercise 6.1 at the end of this chapter).

6.2 A More General Approach

If we regard a collection of two cubes (or other convex polyhedra) as one object, there can be invisible edges that do not belong to backfaces. For example, if one of these cubes is placed behind the other, it can have invisible edges that would be visible if the cube in front were removed. Also, an edge of the cube at the back may be only partly visible. Edges that are partly visible also occur in non-convex polyhedra, as Fig. 6.2 shows. To describe this object, a solid letter L, we use the input format discussed in the previous section:

```
 1   20   0   0
 2   20  50   0
 3    0  50   0
 4    0   0   0
 5   20   0  10
 6   20  40  10
 7    0  40  10
 8    0   0  10
 9   20  40  80
10   20  50  80
11    0  50  80
12    0  40  80

Faces:
1  2  10  9  6  5.
3  4  8  7  12  11.
2  3  11  10.
7  6  9  12.
4  1  5  8.
9  10  11  12.
5  6  7  8.
1  4  3  2.
```

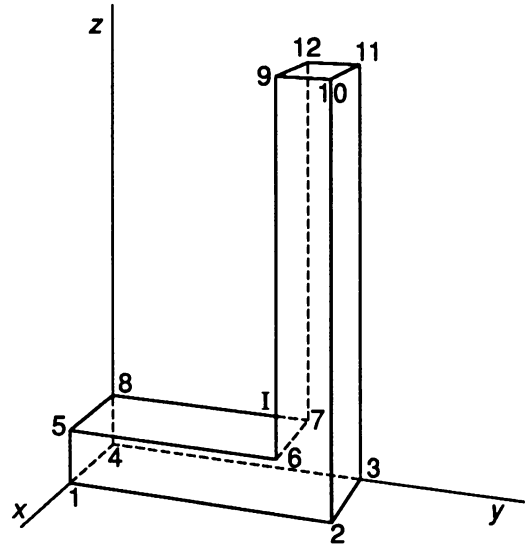


Fig. 6.2. A non-convex polyhedron

Since this polyhedron is not convex, we cannot correctly eliminate all hidden lines by using the method outlined in Section 6.1. Edge (6, 7) is invisible, even though it belongs to the polygon (5, 6, 7, 8), which is not a backface. An interesting edge is (8, 7), which is partly visible. To draw its visible part (8, I), we need to compute I, which is the point of intersection of the line segments (8, 7) and (6, 9) in the image. Both line segments (6, 7) and (I, 7) are obscured by the front face (1, 2, 10, 9, 6, 5).

It will be clear that algorithms for full hidden-line elimination will take much computing time, so any unnecessary work must be avoided. The idea of ignoring backfaces, as discussed in Section 6.1, is therefore very useful. For example, edge (1, 4) belongs only to the backfaces (1, 5, 8, 4) and (1, 4, 3, 2), and therefore need not be stored as a potential line segment to be drawn. Instead of storing polygons, we prefer to store the *triangles* into which they can be decomposed. Each face (that is not a backface), when read in, is used both to store its edges as line segments that are possibly to be drawn, and its triangles, because these may obscure other line segments. This is illustrated in Fig. 6.3 for face (5, 6, 7, 8).

It would not be efficient if, for each vertex, we stored its world-coordinates. Since we read the central object point and the viewpoint first (for example, $x_O = 5$, $y_O = 25$, $z_O = 40$, and $\rho = 400$, $\theta = 20°$, $\varphi = 70°$), we can convert the world coordinates to eye coordinates as soon as they are read. These can then be stored in a *vertex table*, which is an array of structures. Actually, each array element V[i] of the vertex table contains not only the eye coordinates of point i but also a pointer to a contiguous list of vertex numbers, as shown in Fig. 6.4.

In this way we need no separate table of line segments. As you can see, the **connect** member of V[i] is a pointer to a sequence of integers, the first of which indicates how many follow. Each of the integers j that follows in this sequence is

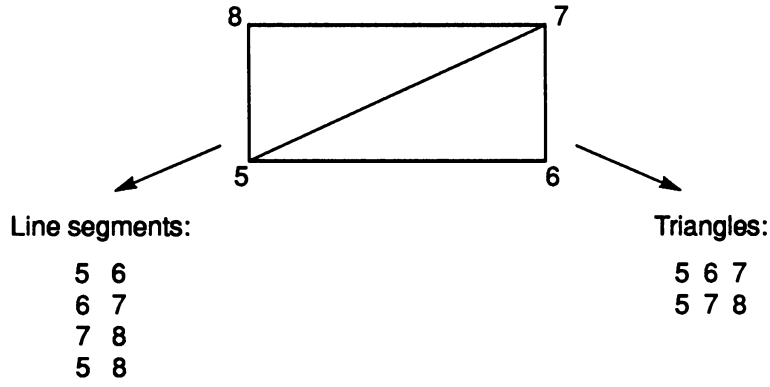


Fig. 6.3. Line segments and triangles derived from a face

greater than i and indicates that there is a line segment (i, j). For example, for vertex 1 we have the line segments (1, 2) and (1, 5). At first sight this may seem to be incomplete, for there is also an edge (1, 4). However, since this is the line of intersection of two backfaces, it is not stored as a candidate to be drawn. Vertex 2 is connected with the vertices 1, 3 and 10, but only the vertex numbers 3 and 10 are stored in the sequence pointed to by **V[2].connect** because edge (1, 2) is already stored in the sequence pointed to by **V[1].connect**. In general, if i is less than j, edge (i, j) is stored in the sequence pointed to by **V[i].connect**, not **V[j].connect**. This is an easy and efficient solution to the problem of double occurrences of line segments, discussed in Section 6.1.

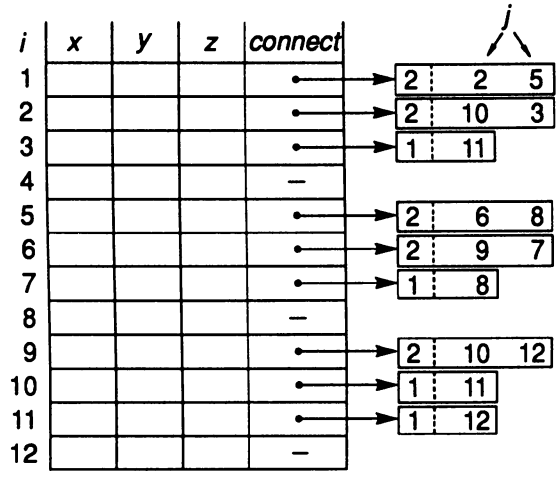


Fig. 6.4. Vertex table V, also representing line segments

Besides vertices and line segments, we must also store (the vertex numbers of) triangles, which is done in the *triangle table*. In each element **triangles[j]** of this table we store the numbers of the three vertices A, B and C, along with a pointer to a structure in which four real numbers a, b, c and h are stored. We will need these four numbers a great many times, and our program would be much slower if we computed them each time. These four numbers are the coefficients that occur in the equation

$$ax + by + cz = h \tag{6.1}$$

which denotes the plane through the points A, B and C. They are stored only once for all triangles that belong to the same polygon, and they are chosen such that

$$a^2 + b^2 + c^2 = 1 \quad \text{and} \quad h \geq 0$$

Then we can write (6.1) as the dot product

$$\mathbf{n} \cdot \mathbf{x} = h \tag{6.2}$$

where

$$\mathbf{n} = [a \quad b \quad c]$$
$$\mathbf{x} = [x \quad y \quad z]$$

The *normal vector* **n** is the vector of unit length that is perpendicular to the plane of the triangle. Equation (6.2) is illustrated by Fig. 6.5, which shows that h is the distance between E and the plane in question. (Recall that E is the origin of the eye-coordinate system that we are using.)

Our task now consists of drawing all line segments PQ (stored as discussed above) as far as they are visible. This means that each such line segment PQ must be examined, using all triangles stored in the triangle table to see if there are some that obscure PQ. Applying this principle literally would lead to a very time-consuming

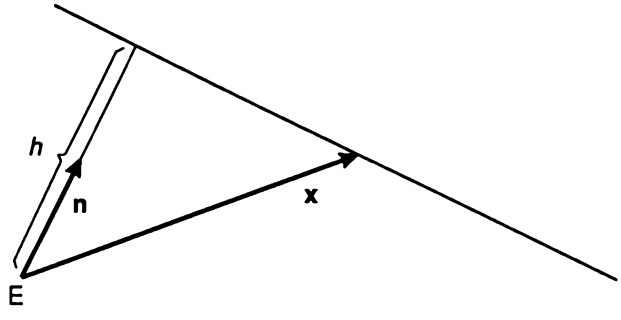


Fig. 6.5. Illustration of $\mathbf{n} \cdot \mathbf{x} = h$

program. For example, if there are a thousand line segments and a thousand triangles, we would need to examine a million pairs, each consisting of a line segment and a triangle. In Section 6.5 we will see that we can do much better by taking only a subset of all triangles into account for each line segment. For the time being, let us assume that we want to draw line segment PQ and that we are given some set of triangles, each of which may completely or partly obscure PQ.

Although each line segment PQ and each triangle ABC is a three-dimensional object, we can deal with their two-dimensional images in many cases. For example, ABC does not obscure PQ if the image of PQ lies outside that of ABC. Instead of computing the screen coordinates each time we need them, we can store them in the vertex table. However, it would be a waste of memory space if we stored the screen coordinates X and Y (see Fig. 5.8) in addition to the eye coordinates x, y and z. We can base our computations on the simple relations

$$X = \frac{x}{z}$$

$$Y = \frac{y}{z}$$

and use $d \,.\, X + x\_center$ and $d \,.\, Y + y\_center$ instead of X and Y in calls to **move** and **draw** (see also Eqs. (5.12) and (5.13) in Section 5.3). Incidentally, we have not yet discussed how to find a suitable value for the screen distance d. This is done by computing all values X and Y as shown above and keeping track of their smallest and largest values. We can then use these to compute d in such a way that the image fits within the screen boundaries. This is similar to what we did in Section 2.6, so we need not discuss this in detail here. Instead of storing all five values x, y, z, X and Y, we will store the screen coordinates X and Y, together with the eye coordinate z. This enables us to deal with image space most of the time, and compute the eye coordinates $x = X \,.\, z$ and $y = Y \,.\, z$ only when we really need them. Since we know the range of X and Y beforehand, we can represent them by integers, ranging, say, from 0 to 32000, by using a scale factor. (This value 32000 was chosen such that, for any C++ implementation, it can be represented by type **int**, while its square does not exceed *LONG_MAX*, the largest possible value of type **long int**, defined in the header file LIMITS.H; this would not necessarily be the case if we wrote *INT_MAX* instead of 32000.) On most computers we can compute much faster with integers than with floating-point numbers, so this really speeds things up. Since it also complicates the program, we will ignore this aspect in our discussion.

6.3 Tests for Visibility

We will use a function, **linesegment**, which is given two points P and Q as arguments. This function's task is to use all triangles of a given set to find out if these obscure part of PQ. If they do not, PQ is drawn; if they obscure PQ completely, no line segment is drawn. Things are more complicated if a triangle obscures only a part of PQ, as we will see in a moment. Let us first discuss the parameters of **linesegment**.

We could make a distinction between a point P in 3-space and its image P′ on the screen. To simplify our discussion, however, we will often simply write P, Q, A, B, C for points on the screen if it is clear from the context that we are actually discussing image points, not object points in 3-space. Thus, if triangle ABC in 3-space lies in front of line segment PQ and partly obscures it, we will say that we compute the points of intersection of PQ and triangle ABC when we are dealing with their images on the screen. For example, P, Q, A, B, C denote image points in Fig. 6.9 but object points in Fig. 6.10. Remember that E is our viewpoint and that the image screen is perpendicular to the z-axis. If PQ lies outside triangle ABC on the screen, as shown in Fig. 6.8(a), then in 3-space PQ lies outside *pyramid* EABC.

For each call to function **linesegment**, a set of relevant triangles is constructed beforehand, as we will discuss in Section 6.5. The triangles we are dealing with are uniquely identified by integers j, which are subscripts for the triangle table. The set of triangles just mentioned is therefore actually a set of triangle numbers j, and will therefore be represented by the following elements of an **int** array **trset**:

trset[0], trset[1],..., trset[ntrset–1]

The call **linesegment(P, Q, ntrset)** then draws any visible portions of PQ. The way this function works is shown below in pseudo code:

```
void linesegment(point &P, point &Q, int k0)
{  int k=k0;
   while (--k >= 0) // k = k0-1, k0-2,...
   {  Perform tests to see if triangle ABC, stored as
      triangles[trset[k]], obscures line segment PQ or
      part of it. There are three cases to be distinguished:
      Case 1. Triangle ABC does not obscure PQ at all.
      Action: Proceed with the next triangle, if any
              (by executing a continue-statement).
      Case 2. Triangle ABC completely obscures PQ.
      Action: Terminate this loop and leave the function
              (by executing a return-statement).
      Case 3. Triangle ABC partly obscures PQ.
      Action: Compute the points of intersection I
              and J (with I nearer to P and J to Q);
              if (P is outside triangle ABC)
                 linesegment(P, I, k);
              if (Q is outside triangle ABC)
                 linesegment(Q, J, k);
              return;
   }
   Draw line segment PQ  // No return-statement was executed,
                         // so PQ is visible
}
```

Remember, **linesegment** is called for every line segment stored as shown in Fig. 6.4. Each call is preceded by constructing a set of triangles, as the following pseudo code shows:

```
for every line segment PQ:
{  Construct a set of triangles that possibly obscure PQ;
   store them in the first ntrset elements of array trset.
   Perform the following function call:
   linesegment(P, Q, ntrset);
}
```

As the pseudo code for function **linesegment** shows, there can be two recursive calls for subsegments of PQ. Clearly, those triangles which do not obscure PQ do not obscure subsegments of PQ either. Only a subset

trset[0], trset[1],..., trset[k0–1]

(where $k0 \leq ntrset$) of the given set of triangles is therefore to be considered in these recursive calls, which explains parameter $k0$.

In the while-loop of function **linesegment** we have to find out whether triangle ABC, identified by integer k, obscures (part of) PQ. This has to be done as fast as possible, which means that we give priority to tests that are likely to succeed and that do not require much (floating-point) computation. The following tests are done in the given order:

Test 1 (3D; Fig. 6.6)

If PQ (in 3-space) is identical with AB, BC or AC, triangle ABC does not obscure PQ. We do not use the coordinates of points P, Q, A, B and C in this test but simply their vertex numbers.

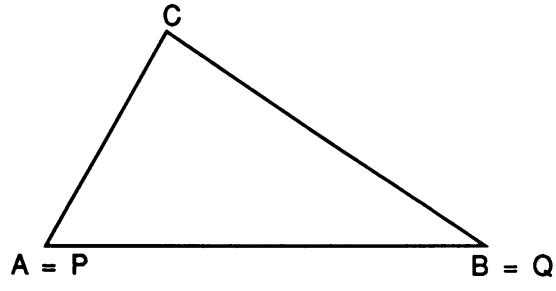


Fig. 6.6. Test 1: PQ identical with AB

Test 2 (2D; Fig. 6.7)

If both P and Q lie to the left of A, B and C, triangle ABC does not obscure PQ. The same is true if both P and Q lie to the right of A, B and C. Since we are comparing

the minimum x-coordinate of one set of points with the maximum x-coordinate of another, we call these tests *minimax tests* in the x-direction. Analogous minimax tests are performed in the y-direction.

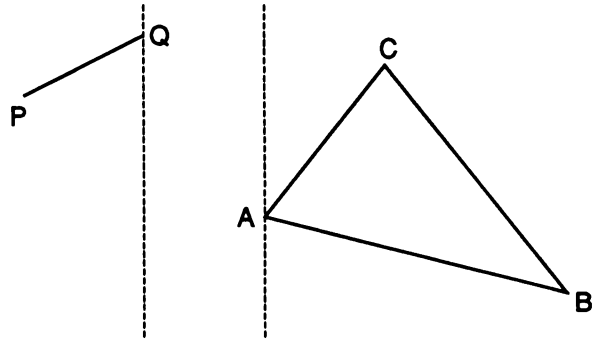


Fig. 6.7. Test 2: Both P and Q to the left of A, B and C

Test 3 (2D; Fig. 6.8)

If P and Q lie outside the triangle and on the same side of the (infinite) line AB, as shown in Fig. 6.8(a), triangle ABC does not obscure PQ. This condition should include the case that either P or Q lies on AB, as shown in Fig. 6.8(b). Similar tests apply to the (infinite) lines BC and CA. Note that we must not include the case that both P and Q lie on a triangle side because that side may be a diagonal of the polygon of which A, B and C are vertices; in that case PQ may lie behind that diagonal in 3-space and therefore be invisible.

We can use the notion of *orientation* in this test. As usual, we define the orientation of the point sequence (P, Q, R) to be 1 if these points, in that order, are counterclockwise, –1 if they are clockwise, and 0 if they lie on a straight line. We know that the orientation of (A, B, C) is positive (otherwise triangle ABC would be a backface). Writing simply o instead of **orientation** (see Section 3.3), we can express this test as

o(A, P, B) + o(A, Q, B) > 0 or
o(B, P, C) + o(B, Q, C) > 0 or
o(C, P, A) + o(C, Q, A) > 0

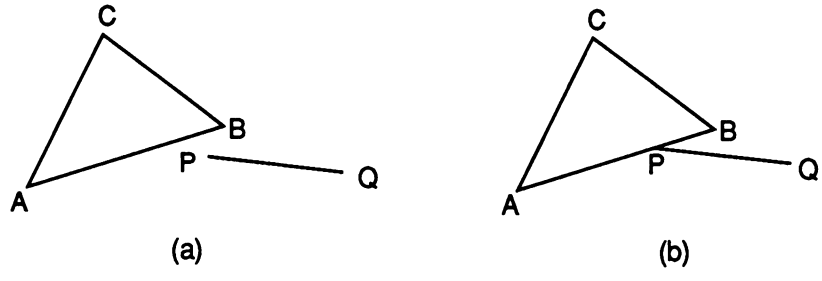


Fig. 6.8. Test 3, using s = o(A, P, B) + o(A, Q, B): (a) *s = 2*; (b) *s = 1*

Test 4 (2D; Fig. 6.9)

If A, B and C lie on the same side of the line through P and Q, triangle ABC does not obscure PQ. This condition should include the case that exactly one of the points A, B and C lie on that line, as shown in Fig. 6.9(b). It can be written as

$$|o(P, Q, A) + o(P, Q, B) + o(P, Q, C)| > 1$$

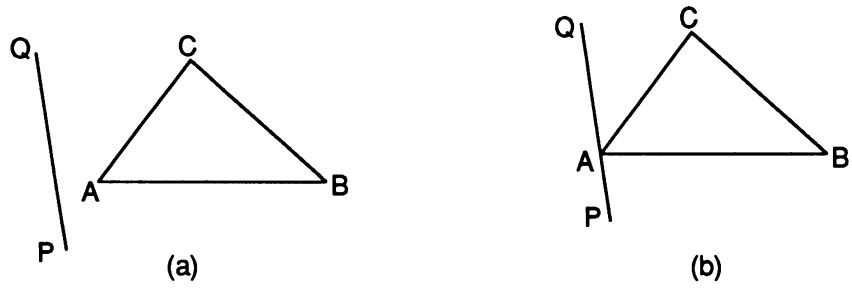


Fig. 6.9. Test 4, using $t = |o(P, Q, A) + o(P, Q, B) + o(P, Q, C)|$: (a) $t = 3$; (b) $t = 2$

Test 5 (3D; Fig. 6.10)

If both z_P and z_Q are less than z_A, z_B and z_C, triangle ABC does not obscure PQ. (This is a minimax test in the z-direction).

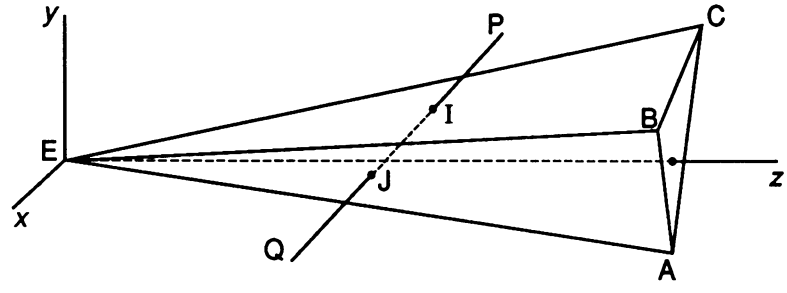


Fig. 6.10. Test 5: z-coordinates of P and Q less than those of A, B and C

Test 6 (3D; Fig. 6.11)

If neither P nor Q lies behind the plane through A, B and C, triangle ABC does not obscure PQ. In this case the points P, Q and E lie on the same side of plane ABC. However, as shown in Fig. 6.11, z_P may be greater than z_A, which was not the case in Test 5. Test 6 is therefore stricter than Test 5 (but is more costly; otherwise Test 5 would be useless).

We now benefit from the fact that we have stored the normal vector $\mathbf{n}$ of plane ABC and the distance h between E and this plane (see Eqs. (6.1) and (6.2)). The condition of this test can be written as

$$\mathbf{n} \cdot \mathbf{EP} \leq h \quad \text{and} \quad \mathbf{n} \cdot \mathbf{EQ} \leq h$$

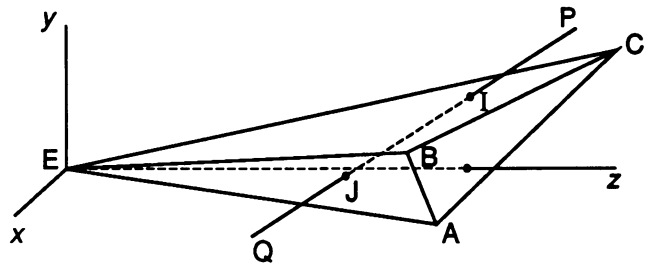


Fig. 6.11. Test 6: Neither P nor Q behind plane ABC

Test 7 (2D; Fig. 6.12)

If neither P nor Q lie outside the triangle (and the previous tests were indecisive), PQ lies behind triangle ABC and is therefore completely invisible. We can again make use of the notion of orientation. For example, P lies outside triangle ABC if at least one of the three orientations (A, P, B), (B, P, C) and (C, P, A) is equal to 1. Although this test is about points in 3-space (PQ lying *behind* triangle ABC), we only have to use screen coordinates in this test, so it is a 2D test from a computational point of view.

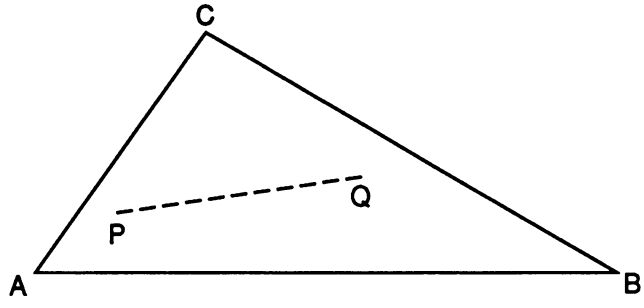


Fig. 6.12. Test 7: PQ behind triangle ABC

In the remaining cases, the relationship between PQ and the current triangle ABC is to be examined in greater detail. If point P lies outside the triangle, there is a point I in which PQ intersects AB, BC, or CA; then subsegment PI lies outside the triangle and has to be drawn. Similarly, if Q lies outside the triangle, subsegment QJ lies outside the triangle and still has to be drawn, where J is a point of intersection of PQ and one of the triangle sides. Note that P and Q may both lie outside the triangle; in that case, we have two points of intersection, I and J, with J lying between I and Q.

Computing these points I and J is no trivial matter. We do not know in advance which of the triangle sides AB, BC and CA give points of intersection with PQ. We solve this problem by trying all three sides. Let us consider only AB, which intersects PQ at point I. We will find I by using the following vectors:

$\mathbf{u} = [u_1 \quad u_2] = \mathbf{PQ}$
$\mathbf{v} = [v_1 \quad v_2] = \mathbf{AB}$
$\mathbf{PI} = \lambda\mathbf{u}$
$\mathbf{AI} = \mu\mathbf{v}$
$\mathbf{w} = [w_1 \quad w_2] = \mathbf{PA}$

We then have to solve the vector equation

$$\mathbf{w} + \mu\mathbf{v} = \lambda\mathbf{u} \tag{6.3}$$

for λ and μ. Figure 6.13 illustrates this vector equation.

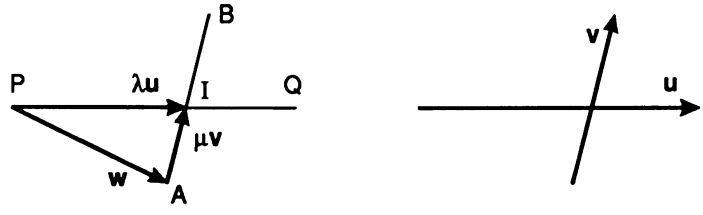


Fig. 6.13. Vector equation $w + \mu v = \lambda u$

If point I really lies between P and Q then λ lies between 0 and 1. Similarly, if it lies between A and B then μ lies between 0 and 1. Solving Eq. (6.3) gives

$$\lambda = \frac{v_2 w_1 - v_1 w_2}{u_1 v_2 - u_2 v_1}$$

$$\mu = \frac{u_2 w_1 - u_1 w_2}{u_1 v_2 - u_2 v_1}$$

In principle, we have to do this computation not only for AB but also for BC and CA, and we are interested only in pairs (λ, μ) that satisfy

$$0 \le \lambda \le 1 \quad \text{and} \quad 0 \le \mu \le 1 \tag{6.4}$$

If exactly one of the points P and Q lies inside triangle ABC, we can stop as soon as we have found such a pair with λ satisfying $0 < \lambda < 1$. Point I is then found by applying the vector equation $\mathbf{PI} = \lambda\,\mathbf{PQ}$. Otherwise we use the minimum and the maximum values λ_{min} and λ_{max} satisfying (6.4) to compute the two points of intersection I and J by using the vector equations

$\mathbf{PI} = \lambda_{min}\,\mathbf{PQ}$
$\mathbf{PJ} = \lambda_{max}\,\mathbf{PQ}$

So much for the computations in image space. However, we want to use points I and J as new point P and Q in recursive calls to **linesegment**, so we also need to know their z-coordinates. Although we have

$$x_I = (1 - \lambda)x_P + \lambda x_Q$$
$$y_I = (1 - \lambda)y_P + \lambda y_Q$$

we cannot use an analogous relationship to compute z_I from the known values z_P and z_Q. For example, if I lies in the middle of PQ in the image, then this is not necessarily the case with the corresponding points in 3-space. (Recall our discussion at the beginning of Section 5.6, illustrated by Figs. 5.19 and 5.20.) Fortunately, we can compute z_I in a surprisingly elegant way by using

$$\frac{1}{z_I} = (1 - \lambda)\frac{1}{z_P} + \lambda\frac{1}{z_Q} \tag{6.5}$$

This equation can be derived using Fig. 6.14, in which we ignore y-coordinates for simplicity.

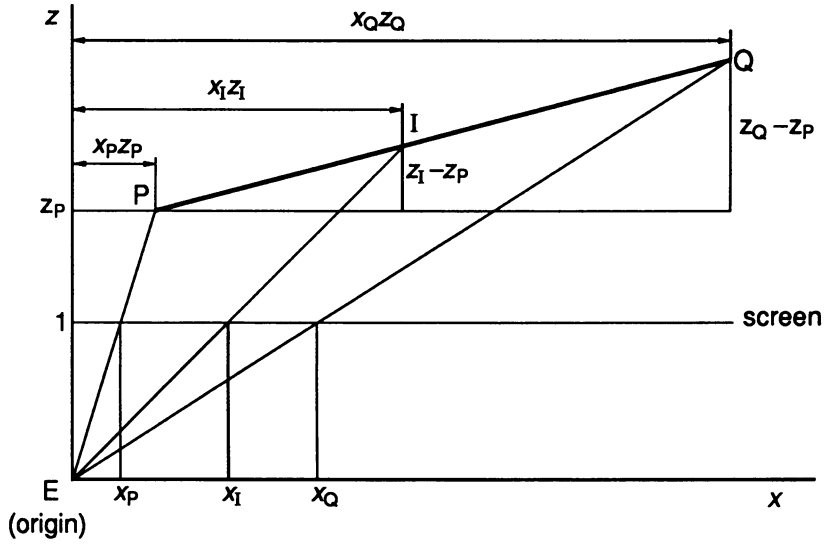


Fig. 6.14. Deriving the z-coordinate of point I

The slope of PQ can be written as the left-hand and the right-hand sides of the following equation:

$$\frac{z_I - z_P}{x_I z_I - x_P z_P} = \frac{z_Q - z_P}{x_Q z_Q - x_P z_P}$$

Multiplying both sides by the product of the two denominators, simplifying the result, and dividing it by $z_I z_P z_Q$ gives

$$\frac{x_Q - x_P}{z_I} = \frac{x_Q - x_I}{z_P} + \frac{x_I - x_P}{z_Q}$$

We then use the substitutions $x_I - x_P = \lambda(x_Q - x_P)$ and $x_Q - x_I = (1 - \lambda)(x_Q - x_P)$ and divide the resulting equation by $x_Q - x_P$ to obtain Eq. (6.5).

Test 8 (Fig. 6.15)

If point I or point J lies in front of the triangle, triangle ABC does not obscure PQ. In contrast to Test 6, point Q, for example, may lie behind plane ABC, as Fig. 6.15 illustrates. (Since we do not allow line segments to penetrate faces, it is not possible for I and J to lie on different sides of plane ABC.)

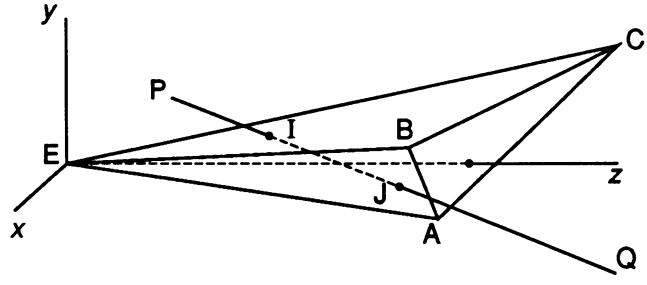


Fig. 6.15. Test 8: Neither I nor J behind plane ABC

Recursion elimination

Using recursion, as suggested in the above discussion, may cause stack overflow. The risk of this is real here since we cannot predict the maximum recursion depth that may occur. We will therefore actually eliminate recursive calls by means of a linked list, used as a stack. Instead of calling **linesegment** recursively, we store the arguments that would occur in such calls on this stack. In the **main** function we repeatedly take elements from this stack and perform the corresponding calls to **linesegment** until the stack is empty.

6.4 Holes; Loose Line Segments and Planes

Holes

The way we specify the boundary faces of objects as discussed so far does not work for polygons that have holes in them. For example, consider the solid letter A of Fig. 6.16, the front face of which is not a proper polygon because there is a triangular hole in it. The same applies to the (identical) face at the back. Each vertex i of the front face is connected with vertex $i + 10$ of the back face ($1 \leq i \leq 10$).

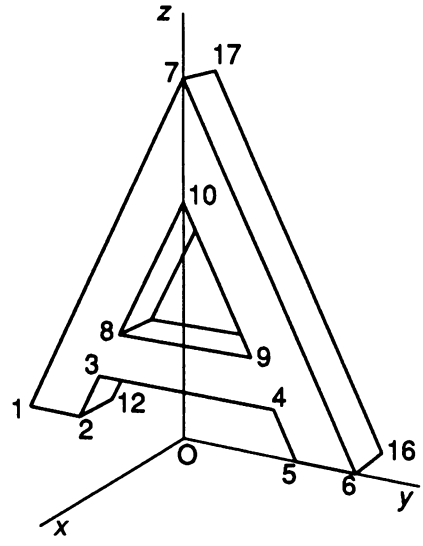


Fig. 6.16. Solid letter A

We can turn the front face into a polygon by introducing a very narrow gap, say, between the vertices 7 and 10, as shown in Fig. 6.17. After doing this, we could specify this new polygon as

```
1  2  3  4  5  6  7  10  9  8  10  7.
```

Note that this can be done only if the gap (7, 10) has width zero, so that there is only one vertex (7) at the top. On the other hand, the ordering of the vertex numbers 10, 9, 8 (rather than 10, 8, 9) in this list is logical only if we consider this gap to be real, as shown in Fig. 6.17. If we really specified the front face by the above sequence of vertex numbers, the line (7, 10) would be regarded as a polygon edge and therefore appear in our picture. This is clearly undesirable. To prevent this from happening, we introduce a code, for which we use minus signs, as in

```
1  2  3  4  5  6  7  -10  9  8  10  -7.
```

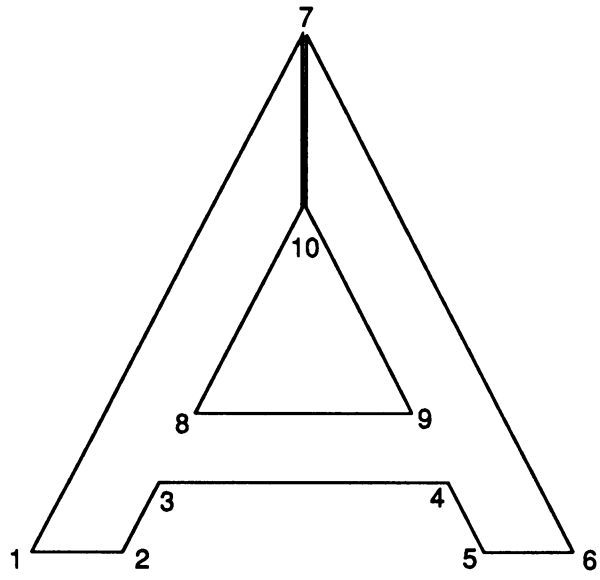


Fig. 6.17. A polygon

In general, with two vertex numbers m and n, the line segment (m, n) will not be drawn if we write

$m \ -n$

in the input data. This principle is applied twice in the above list, namely for the line segments (7, –10) and (10, –7). The solid letter A of Fig. 6.16 can therefore be obtained by using the following input file:

```
1 0 -30 0
2 0 -20 0
3 0 -16 8
4 0 16 8
5 0 20 0
6 0 30 0
7 0 0 60
8 0 -12 16
9 0 12 16
10 0 0 40
11 -10 -30 0
12 -10 -20 0
13 -10 -16 8
14 -10 16 8
15 -10 20 0
```

```
16 -10 30 0
17 -10 0 60
18 -10 -12 16
19 -10 12 16
20 -10 0 40
Faces:
1 2 3 4 5 6 7 -10 9 8 10 -7.
11 17 -20 18 19 20 -17 16 15 14 13 12.
2 12 13 3.
3 13 14 4.
15 5 4 14.
8 9 19 18.
8 18 20 10.
19 9 10 20.
6 16 17 7.
11 1 7 17.
11 12 2 1.
15 16 6 5.
```

(Note that this use of minus signs applies only to vertex numbers, which normally are positive integers; minus signs used for coordinates have their usual meaning.) Implementing this idea in program HIDELINE (listed in Appendix A) is very simple. For example, because of the minus sign that precedes vertex number 10 in

```
... 7 -10 ...
```

we do not store line segment (7, 10) in the linked lists pointed to in the vertex table (as shown in Fig. 6.4). In other respects, we simply ignore these minus signs, so the set of triangles resulting from the above input data is the same as the set we would have if the minus signs had been omitted.

Loose line segments

It is sometimes desirable to include line segments other than polygon edges. We will simply specify such a line segment by a number pair, followed by a period. Examples of such 'loose' line segments are the x-, y- and z-axes in Fig. 6.16. To allow line segments that are not polygon edges, we accept 'face specifications' of only two vertex numbers and interpret these as line segments. For example, we can introduce the coordinate axes just mentioned by inserting the input lines

```
21    0    0    0
22   40    0    0
23    0   40    0
24    0    0   70
```

before the keyword **Faces**, and the following lines after it:

```
21  22.
21  23.
21  24.
```

These number pairs do not define polygons, so they do not contribute to our set of triangles. They are stored as line segments, however. It will now be clear why we have frequently used the more general term *line segment* rather than *edge*.

Loose planes

Although our polygons are normally boundary faces of solid objects, we sometimes want to consider very thin (finite) planes by themselves. Examples are sheets of paper, such as the pages of a book, and a cube made of very thin material, of which the top face is omitted, as shown in Fig. 6.18.

Since such thin planes are in principle visible from both sides, we have to specify each side, which gives two different point sequences because the orientation of the point sequence must be counter-clockwise. For example, the front face of the cube in Fig. 6.18, with vertex numbers 1, 2, 3, 4, was specified twice in the input file:

```
1  2  3  4.
4  3  2  1.
```

The second of these two lines is not strictly necessary for the image shown in the figure, which was produced using $\rho = 5$, $\theta = 30°$, $\varphi = 75°$, but we really need that line to obtain the same image when we view the cube from the back, using $\rho = 5$, $\theta = 210°$ and $\varphi = 75°$, for example.

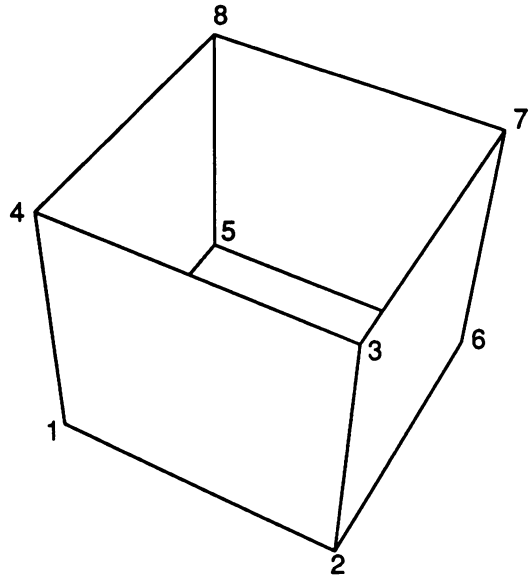


Fig. 6.18. A hollow cube

6.5 Reducing the Number of Visibility Tests

As pointed out in Section 6.2, our hidden-line algorithm would be rather slow if we really compared each line segment with all triangles that we have stored (in array **triangles**). Fortunately, we can reduce the number of such comparisons significantly. For example, when we are investigating the visibility of line segment PQ that lies near the top left corner of the screen, we can ignore triangles that lie near the bottom right corner. As Fig. 6.19 illustrates, we divide the screen into 8 × 8 (or, in general, into *Nscreen* × *Nscreen*) rectangles of equal size.

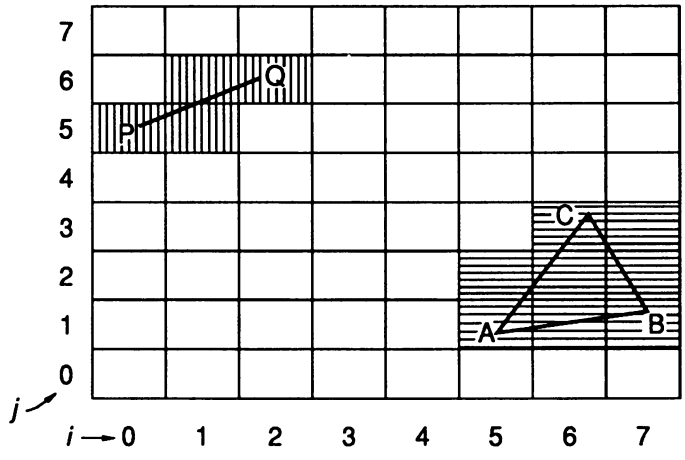


Fig. 6.19. A screen divided into 8 × 8 rectangles

Our improvement in efficiency is based on the idea of setting up a linked list of triangles for each of these 64 rectangles. For each rectangle, only the triangles that have points in common with it are stored in its linked list. We will say that such triangles are *associated* with the rectangle in question, and we will refer to these linked lists as *screen lists*. Since we have already stored all triangles in array **triangles** and we do not want to duplicate data, each node of a screen list need contain only an integer subscript value of this array and a pointer to the next element of the list, if any. The start pointer for the rectangle in column *i* and row *j* (where *i* and *j* range from 0 to *Nscreen*–1) is **SCREEN[i][j]**, where **SCREEN** is declared as

```
struct node {int jtr; node *next;} *SCREEN[Nscreen][Nscreen];
```

The **int** member *jtr* is used as an index for array **triangles**. For example, the following program fragment shows how we can use all triangles associated with the rectangle in column *i* and row *j*:

```
node *p;
p = SCREEN[i][j];
while (p != NULL)
{  ...     // Use triangle[p->jtr]
   p = p->next;
}
```

For each line segment PQ that is to be tested for visibility, we construct a set of relevant triangles. Initially, this set is empty. Then for each rectangle that has points in common with PQ, we add its associated triangles to the set. In this way the rather time-consuming tests for visibility, discussed in Section 6.3, are restricted to relatively few triangles, which speeds up our program considerably.

A complete discussion of program HIDELINE would be very tedious; those who are interested in further programming details are therefore referred to Appendix A.

Exercises

6.1 Write a program for hidden-line elimination, restricted to convex polyhedra, as outlined in Section 6.1.

6.2 Write a program that generates an input file for program HIDELINE, so that this can draw a book with all pages partly visible, as shown in Fig. 6.20. The input data of your program consists of the height and the width of a page and the number of pages, n. Use $180°/(n - 1)$ as the angle between two successive pages. Note that each page has two sides that in principle can be visible; refer to our discussion of 'loose planes', at the end of Section 6.4.

(More exercises of this type, as well as similar programming examples, can be found in Chapter 8.)

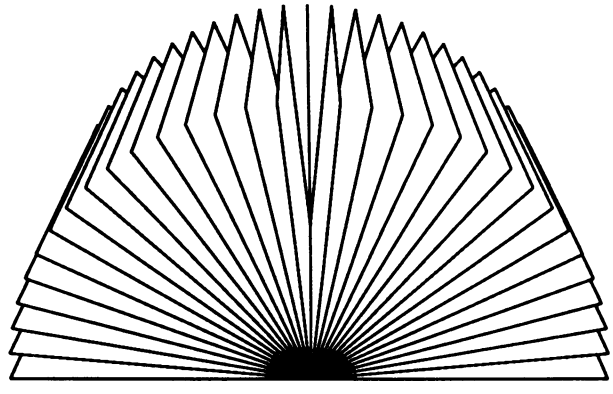


Fig. 6.20. A book (Exercise 6.2)

7

Hidden-surface Elimination

7.1 Colors and Palettes Applied to 3D Faces

So far, we have represented three-dimensional objects by line drawings. These are similar to the work of an artist who draws with a pen or pencil. By contrast, the pictures to be discussed in this chapter are more similar to those produced by *painters*, who normally use brushes to paint in colors. Such pictures can be produced by program HIDEFACE, listed in Appendix B. The principles on which this program is based form the subject of this chapter. Again, the program is easy to use. It can read the same input files as those discussed in Chapter 6, so these files can lead to two types of graphics output. (There is one exception, however: 'loose line segments', as discussed in Section 6.4, are ignored by program HIDEFACE because they do not form surfaces.)

We will focus our attention on surfaces consisting of polygons; their images on the screen are also polygons, which we can fill with given colors, as discussed in Chapter 4. On the lowest level, we will use integer non-negative coordinates X and Y, with maximum values $X\_\_max$ and $Y\_\_max$ and with the origin in the lower-left corner of the screen (see Fig. 4.1). Recall that these maximum values are declared in header file GRSYS.H and available after a call to **initgr**. We will assume that the number of available colors will be *ncolors*, which will be equal to 16 in our discussion below. These colors will be identified by the integers 0, 1,..., *ncolors*–1. In Section 4.1, the default meanings of these color numbers for VGA are listed. However, these are not very useful for our present purposes, since what we want is both a sequence C_1, C_2,..., $C_{ncolors-1}$ of colors ranging from dark to light and a background color C_0, which is essentially different from the others. Fortunately, there are computer systems that enable us to define colors ourselves. Even with a limited number of such colors, we can obtain reasonable results.

Color shades

In many three-dimensional applications we want to use at least one foreground color in a number of different shades, ranging from very dark to very light, and exactly one background color, which should be essentially different. We will therefore use a facility consisting of the following very simple function call:

```
shaded_colors();
```

This function is available in our module GRSYS. By default, it defines color 0 as blue, to be used as the background color, and colors 1,..., *ncolors* – 1 as a range of yellow shades, with very dark yellow as color 1 and very light (or bright) yellow as color *ncolors* – 1. After a call to this function immediately after calling **initgr**, the entire screen is blue. Then, before other graphics output, we can call **set_color(i)**, with *i* greater than 0 and less than *ncolors*. Any subsequent graphics output will then appear in a shade of yellow, the brightness of which depends on *i*. For example, the following program draws a rather dark yellow line, followed by a lighter one:

```
// SHADES: Shaded colors.
//         To be linked with GRSYS
#include "grsys.h"

int main()
{  initgr();
   shaded_colors();
   set_color(4);  horline(10, 300, 200);
   set_color(12); horline(10, 300, 100);
   endgr();
   return 0;
}
```

There is another interesting possibility of function **shaded_colors**: instead of using the default colors just mentioned, it can read color definitions from an input file. The above discussion applies to the case that there is no file PALETTE.TXT in the current directory. If there is one, with appropriate contents as we will discuss shortly, the colors defined in this file will be used instead of blue and shades of yellow.

The RGB palette

To make our discussion as concrete as possible we will consider the way colors can be defined with the standard VGA adapter used in the IBM PC and compatible machines. With this color system we can, in principle, define an enormous number of colors, but only *ncolors* = 16 of these can be used at the same time. Although this sounds rather disappointing, we can produce interesting color pictures with these sixteen colors, provided that we define them in a sensible way. This is illustrated by

Plates 3,..., 8. We will define a *palette* by means of the function **set_rgb_palette**[1], declared in GRSYS.H as

```
void set_rgb_palette(int colornr, int R, int G, int B);
```

where each parameter is a non-negative integer: *colornr* is less than *ncolors* (= 16), while *R*, *G* and *B* are less than 64. These last three parameters express the *intensity* (or brightness) of the red, green and blue components in the color. For example, if we want color 6 to be green with maximum intensity, we can write

```
set_rgb_palette(6, 0, 63, 0)
```

The lower the intensity, the darker the color. We can also mix red, green and blue to obtain other colors. Some important special cases are shown in the following table, where *x* denotes some positive value less than 64, expressing the intensity of the color in question:

R	*G*	*B*	
0	0	0	Black
0	0	*x*	Blue
0	*x*	0	Green
0	*x*	*x*	Cyan
x	0	0	Red
x	0	*x*	Magenta
x	*x*	0	Yellow
x	*x*	*x*	Gray (or white if $x = 63$)

Note that $R = G = 0$, $B = 63$ denotes a very bright but rather dark blue; for a lighter shade of blue, we must not reduce the value of *B* but, instead, increase the value of *R* and *G* by some value less than 63. Since red, green and blue give white, this has the effect of adding white to blue, which explains that we obtain a lighter shade of blue.

In program HIDEFACE we will use function **shaded_colors**, mentioned above. Normally, this means that there is a blue background color and fifteen different shades of yellow as foreground colors. However, we can change this by providing a file named PALETTE.TXT. Function **shaded_colors** checks whether there is such a file in the current directory. If so, it must contain sixteen (or, in general, *ncolors*) lines of text of three integers each; the first line consists of the *R*, *G* and *B* values for color 0, the second those for color 1, and so on. If less than *ncolors* × 3 such integers can be read, the program will terminate with an error message. Function **shaded_colors** is listed below:

---

[1] This function is similar to the slightly more complicated Borland C++ function **setrgbpalette** (written without underscores). Appendix C shows how **set_rgb_palette** can be expressed in terms of **setrgbpalette**.

```
void shaded_colors(void)
{  int i, j, red, green, blue;
   ifstream palfile("palette.txt");
   if (palfile) // Is there a palette file?
   {  // If so, use it:
      for (i=0; i<ncolors; i++)
      {  palfile >> red >> green >> blue;
         if (palfile.fail())
            errmess("Incorrect file PALETTE.TXT");
         set_rgb_palette(i, red, green, blue);
      }
   }  else // There is no file PALETTE.TXT; use
          // blue background and yellow foreground as default
   {  set_rgb_palette(0, 0, 0, 63);  // 0 = dark blue
      for (i=1; i<ncolors; i++)
      {  j = 4 * i + 3;
         set_rgb_palette(i, j, j, 0);
         // Shades of yellow: 1 = very dark, 15 = very bright
      }
   }
}
```

As we have already seen in program SHADES, this function is immediately available in module GRSYS.

Here is an example of a file PALETTE.TXT, based on *ncolors* = 16, which defines a green background color and fifteen shades of red for the foreground:

```
0  63  0
4   0  0
8   0  0
12  0  0
16  0  0
20  0  0
24  0  0
28  0  0
32  0  0
36  0  0
40  0  0
44  0  0
48  0  0
52  0  0
56  0  0
60  0  0
```

Replacing the first line with **30 63 30**, for example, would result in a lighter shade of green (green mixed with white) for the background.

Light vector, faces, and color intensities

As in Chapters 5 and 6, we will use a left-handed eye-coordinate system, with its origin coinciding with viewpoint E, and its positive x_e-, y_e- and z_e-axes pointing to the right, upwards, and in the viewing direction, respectively (see Fig. 5.4). Recall that the x_e- and y_e-axes are parallel to the X- and Y-axes of the screen-coordinate system (see Fig. 5.8).

Suppose there is light coming from a far-away source, the sun, for example. Then the direction of the rays of light will be given as a *light vector*. For example, with

light vector = (4, –2, 5)

the light, on going four units in the positive x_e-direction, also goes two units in the negative y_e-direction and five units in the positive z_e-direction. In other words, the light comes from the left (component 4), from above (component –2), and from behind (component 5). We will use this given light vector in combination with the normal vectors of the faces that we want to display, by computing the dot product of these vectors. Let us use normal vectors pointing *towards* the faces. The higher this dot product, the more light the face in question receives per square inch. Consequently, the maximum brightness of the (yellow) color is used when this dot product assumes its maximum value. Similarly, the darkest possible shade of yellow must correspond to the minimum value of this dot product.

To understand the way this idea is implemented, let us first have a look at the data structures that are used:

```
#include "d3.h"  // See Section 5.3
...
int ntr; // Number of triangles
struct triadata {vec3 normal; float h; int color;};
struct tria {int Anr, Bnr, Cnr; int Z; triadata *ptria;}
   *triangles, *ptriangle;
```

Again, we are dealing with objects, bounded by flat faces, which are polygons (possibly containing holes). These polygons are decomposed into triangles, which are stored as

```
triangles[0],..., triangles[ntr-1]
```

Each array element **triangles[i]** contains a pointer **ptria**, pointing to an object of type **triadata**, which contains the normal vector of triangle *i*. The *h* component of this object (not needed for our present purposes) indicates how far the plane of this triangle is away from the viewpoint, measured in the direction of the normal vector. There is also an **int** member *color*, which is the color number for the polygon in question. Note that triangles belonging to the same polygon can share this **triadata** object, because they have the same normal vector, the same distance *h* and the same color.

To be able to map all possible dot products (mentioned above) onto the available color range (1,..., *ncolors*–1), we compute them first for all triangles for the sole purpose of finding their range. This is done by means of the for-statement below:

```
const double BIG=1e30;
double rcolormin=BIG, rcolormax=-BIG, delta;

void findrange(int i)
{  vec3 normal = triangles[i].ptria->normal;
   float rcolor;
   rcolor = dotproduct(normal, lightvector);
   if (rcolor < rcolormin) rcolormin = rcolor;
   if (rcolor > rcolormax) rcolormax = rcolor;
}
...
for (i=ntr-1; i>=0; i--) findrange(i);
delta = 0.999 * (ncolors - 1)/(rcolormax - rcolormin + 0.001);
for (i=ntr-1; i>=0; i--) set_tr_color(i);
```

The value of *delta* is used in function **set_tr_color**, which assigns proper values to the *color* members of all **triangle[i].ptria** objects:

```
void set_tr_color(int i)
{  vec3 normal = triangles[i].ptria->normal;
   int color;
   float rcolor;
   rcolor = dotproduct(normal, lightvector);
   color = 1 + (rcolor - rcolormin) * delta;
   triangles[i].ptria->color = color;
}
```

It follows from the above code that all color values 1,..., *ncolors* – 1 (and no others) will actually be used, regardless of the choice of the light vector. For example, if *rcolor* is equal to *rcolormin*, the color number as computed above will be *color* = **int(1+0.0)** = 1. Similarly, if *rcolor* assumes its maximum value *rcolormax*, then, before truncating to type **int**, the color number is computed as

$$1 + (rcolormax - rcolormin) \cdot delta$$

$$< 1 + (rcolormax - rcolormin) \cdot (0.999 \cdot \frac{ncolors - 1}{rcolormax - rcolormin})$$

$$< 1 + (ncolors - 1)$$

where < should be read as 'slightly less than', so that after truncating to type **int** the value of *color* will in this case be equal to *ncolors* – 1. (Note that the constant 0.001 in the expression for *delta* prevents division by zero in case *rcolormax* = *rcolormin*.)

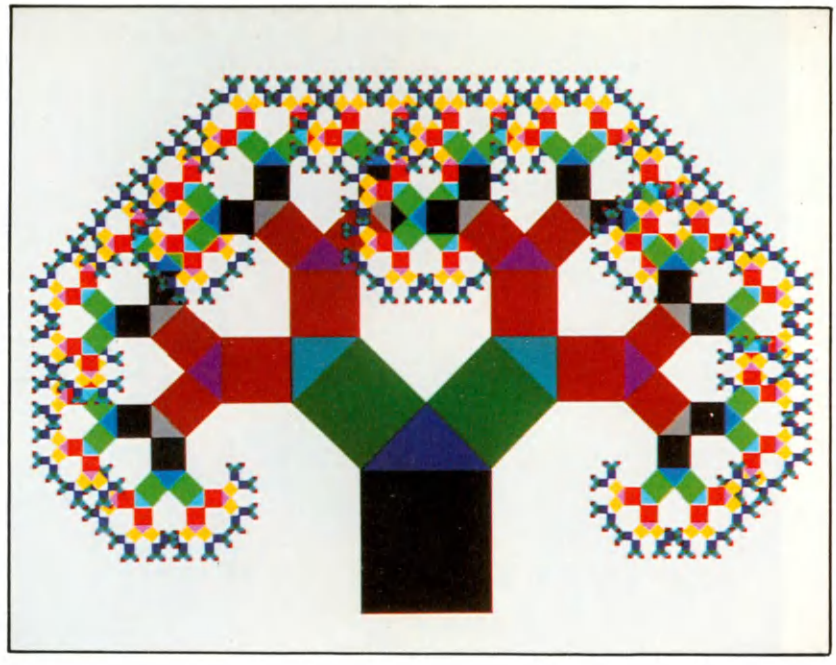


Plate 1 *Asymmetric tree based on depth-first traversal (Exercise 4.4)*

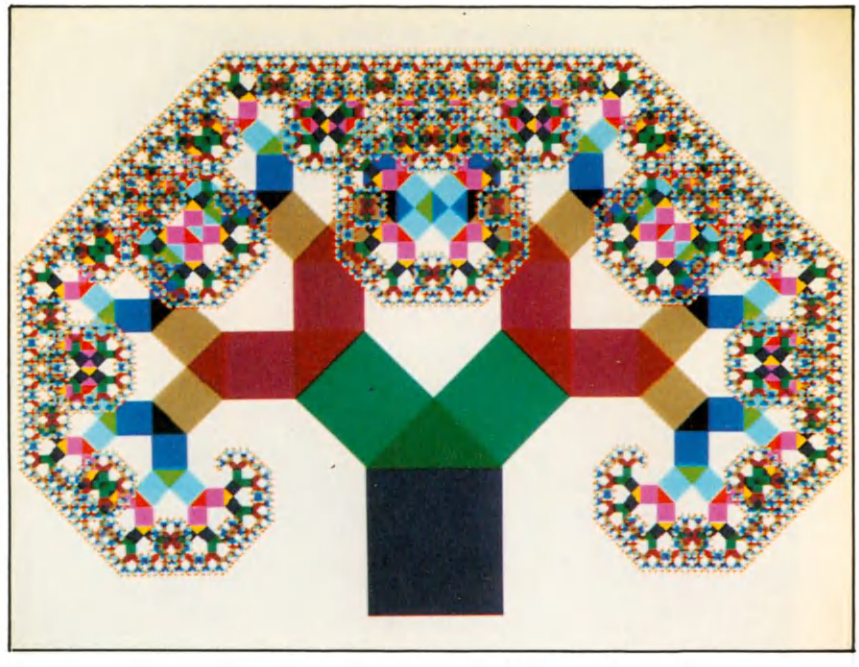


Plate 2 *Tree based on breadth-first traversal (also Exercise 4.4)*

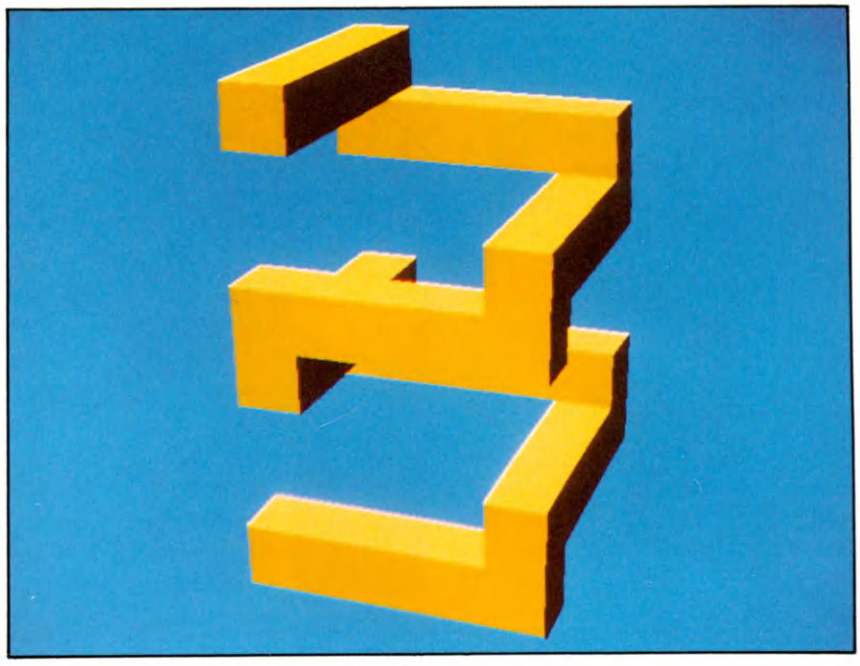


Plate 3 *Eight beams (Section 8.3)*

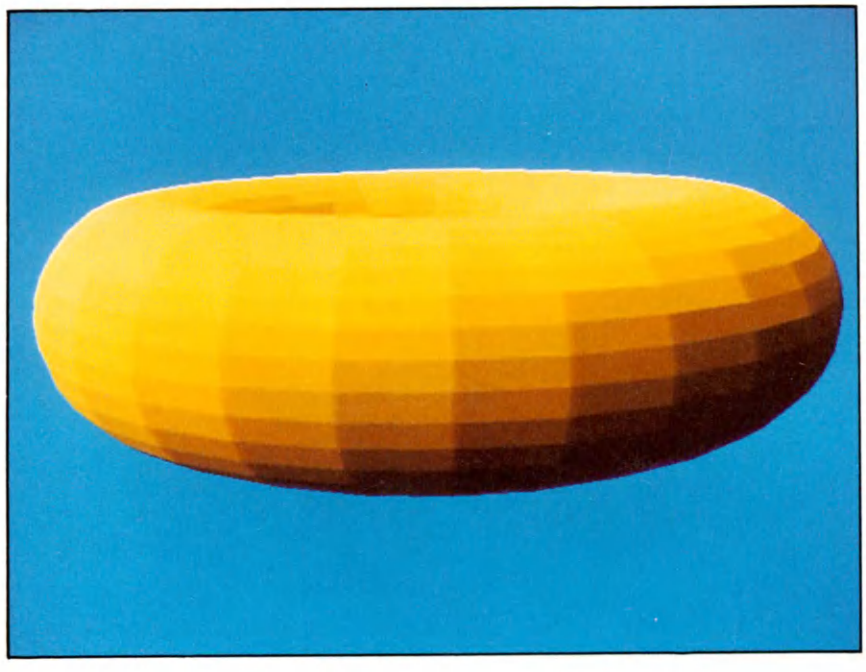


Plate 4 *Torus (Section 8.5)*

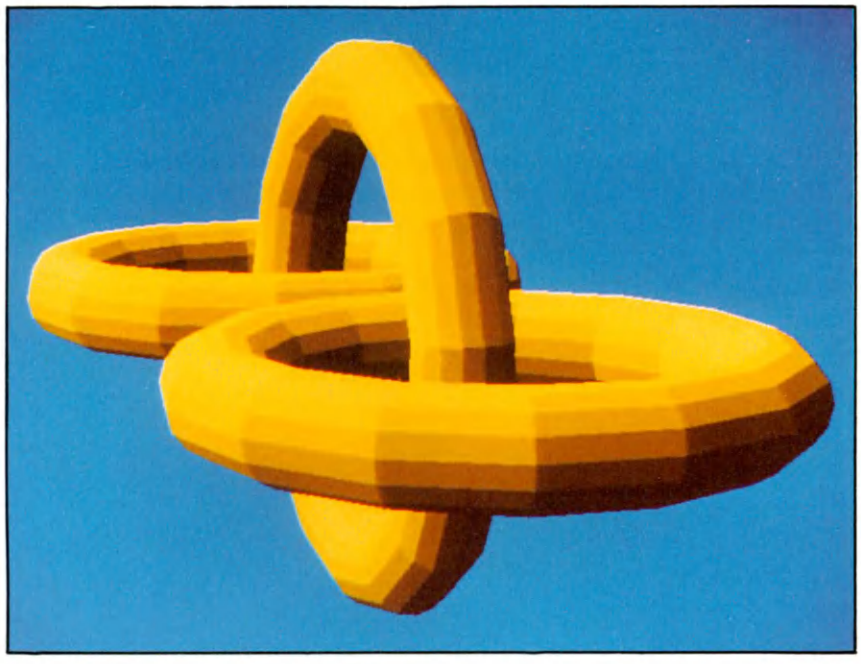


Plate 5 *Three interlocking rings (Exercise 8.8)*

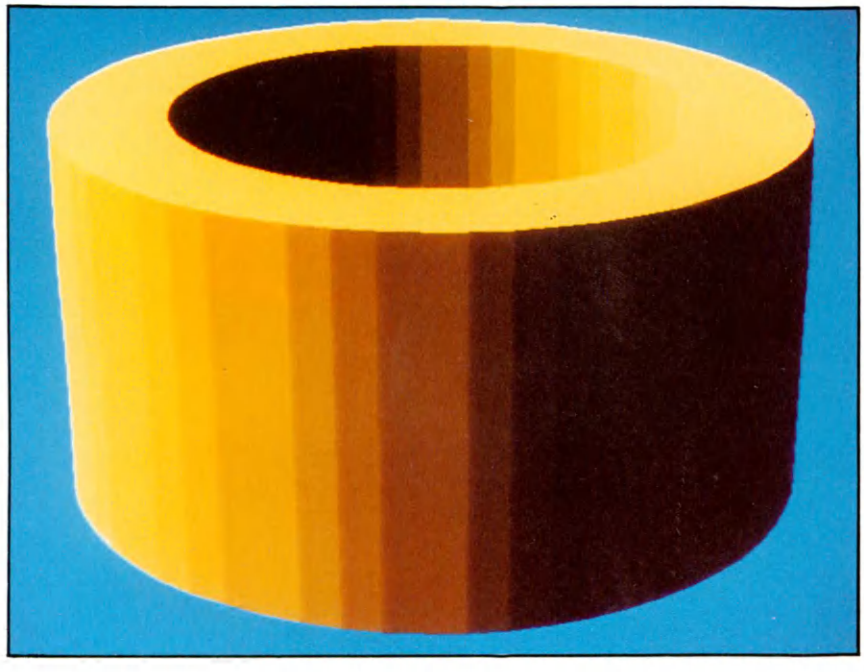


Plate 6 *Hollow cylinder (Section 8.2)*

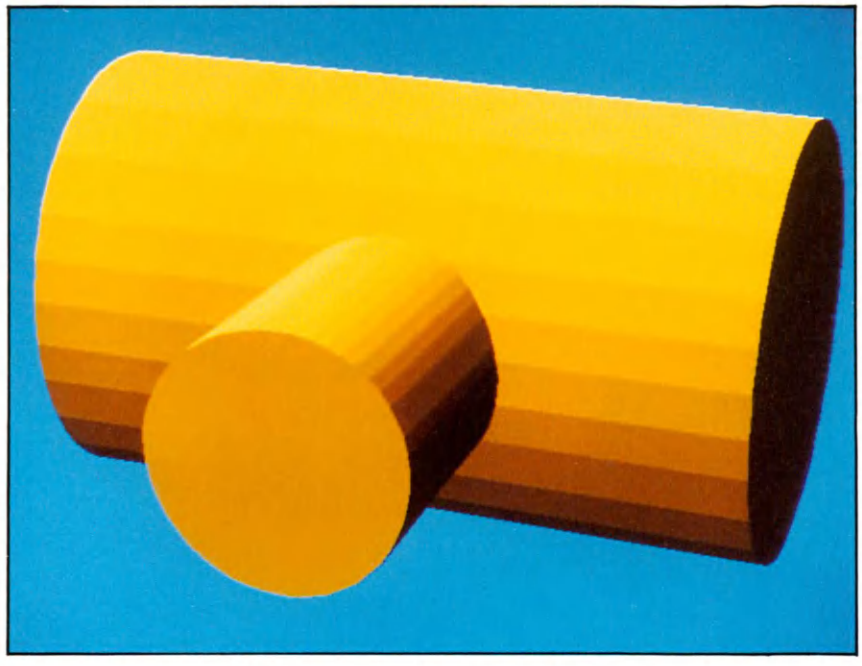


Plate 7 *Two cylinders in a T-form (Exercise 8.10)*

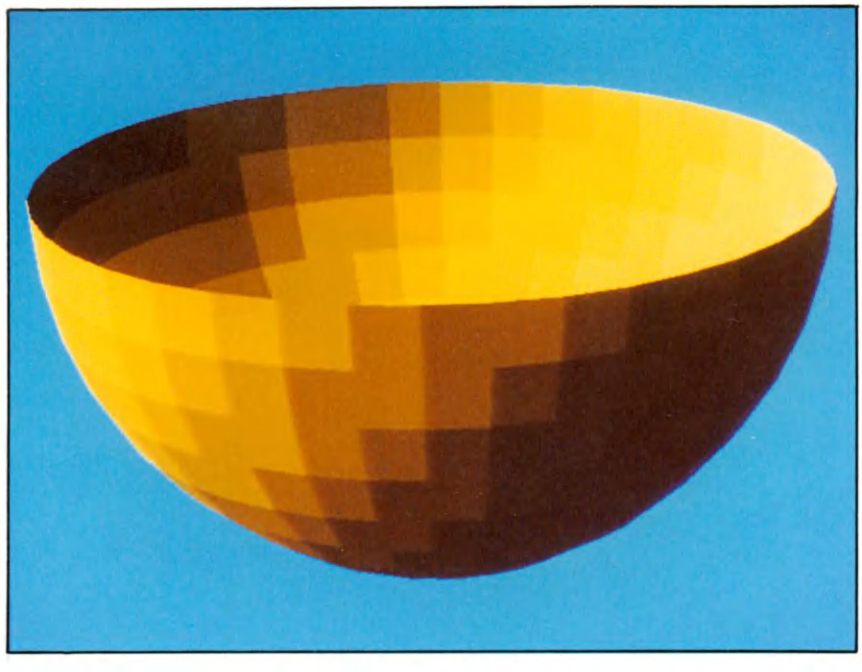


Plate 8 *Semi-sphere (Section 8.6)*

We can later retrieve the *color* members stored above, to use them as arguments in calls to **set_color**, as is done in

```
void fill_window(int i, int Xmin, int Xmax, int Ymin, int Ymax)
{  // Fill entire window with color determined by normal vector
   // of triangle i
   int y;
   set_color(triangles[i].ptria->color);
   Xmin = to_pix(Xmin); Xmax = to_pix(Xmax);
   Ymin = to_pix(Ymin); Ymax = to_pix(Ymax);
   for (y=Ymin; y<=Ymax; y++) horline(Xmin, Xmax, y);
}
```

7.2 Real and Integer Coordinates

Before we proceed with discussing hidden-face algorithms, we must pay some attention to the coordinates that are used in program HIDEFACE. Altogether there are no less than five sorts of them, summarized below:

(1) World coordinates x_w, y_w, z_w. These are the coordinates given by the user reduced by the coordinates x_O, y_O, z_O of the central object point, which may also be supplied by the user. (A computed default point may be used instead.)

(2) Eye coordinates x_e, y_e, z_e. These are computed by function **eyecoord**. They depend on the viewpoint E and are the result of the viewing transformation (see Section 5.2), applied to the world coordinates.

(3) Screen coordinates, which we will now denote as x_s and y_s. They are obtained by applying the perspective transformation (see Section 5.3) to the eye coordinates:

$$x_s = \frac{x_e}{z_e} \qquad y_s = \frac{y_e}{z_e}$$

The three kinds of coordinates just mentioned are real numbers, that is, we use floating-point type for them. Note that they are all denoted by lower-case letters. By contrast, we also use two kinds of coordinates which are integers and denoted by capital letters:

(4) 'Large coordinates', X, Y, and Z, discussed below.

(5) Pixel coordinates X_{pix} lying in the range $0 - X\_\_max$, and Y_{pix} in the range $0 - Y\_\_max$ (see Section 4.1).

It will be clear that pixel coordinates are required for the actual area-filling process. Since computing is much more efficiently done with integers than with floating-point numbers, it is tempting to use pixel coordinates X_{pix} and Y_{pix} instead of screen coordinates x_s and y_s for two-dimensional computations (in the image plane). However,

they may be rather inaccurate; typical values for *X__max* and *Y__max* are 639 and 479, respectively. Instead, we had better use a much larger range of (non-negative) integer coordinates X and Y, which we eventually divide by a constant factor k to obtain X_{pix} and Y_{pix}. This explains the use of the 'large coordinates' X and Y, which will lie in the range 0,..., 32000[1].

The meaning of Z is slightly different in that it is directly related to the eye coordinate z_e. While computing all eye and screen coordinates, we can easily compute the minimum and maximum values of z_e, written as *zemin* and *zemax* in the program. Similarly, *xsmin*, *xsmax*, *ysmin* and *ysmax* are used for the extreme values of x_s and y_s. Then we can compute some very useful values as follows:

```
double xsmin, xsmax, ysmin, ysmax, xsrange, ysrange, xsC, ysC,
       f, fx, fy, xsC, ysC, XLCreal, YLCreal, zfactor;
int k, k1, k2, hk, XLC, YLC, XLmax, YLmax;
const int LARGE=32000;

... // Compute xsmin, xsmax, ysmin, ysmax, zemin, zemax

xsrange = xsmax - xsmin; ysrange = ysmax - ysmin;
xsC = 0.5 * (xsmin + xsmax);
ysC = 0.5 * (ysmin + ysmax);
k1 = LARGE/(X__max+1); k2 = LARGE/(Y__max+1);
k = min2(k1, k2); // k is the smaller of k1 and k2
hk = k/2;
XLmax = k * (X__max+1); YLmax = k * (Y__max+1);
// Pixel coordinates will be Xpix = to_pix(X) and Ypix = to_pix(Y)
XLC = XLmax/2; YLC = YLmax/2;
XLCreal = XLC + 0.5; YLCreal = YLC + 0.5;
fx = XLmax/xsrange; fy = YLmax/ysrange;
f = 0.95 * (fx < fy ? fx : fy);
zfactor = LARGE/(zemax - zemin);
```

The values *f*, *xsC*, *ysC*, *XLCreal* and *YLCreal*, computed above, are used to derive the large coordinates X and Y from the screen coordinates x_s and y_s, by means of the following functions:

```
inline int XLarge(double xs) {return int(XLCreal + f * (xs-xsC));}
inline int YLarge(double ys) {return int(YLCreal + f * (ys-ysC));}
```

Similarly, the values *zemin* and *zfactor* are used to compute the large coordinate Z when the corresponding eye coordinate z_e is given:

---

[1] *INT_MAX* and *LONG_MAX* (defined in LIMITS.H) are at least equal to 32767 and 2147483647, respectively. The range chosen here therefore enables us to represent not only X and Y by type **int** but also their squares by type **long int**, which would not necessarily have been the case if we had used *INT_MAX* instead of 32000.)

```
inline int ZLarge(double ze) {return int((ze-zemin)*zfactor+0.5;)}
```

We need the inverses of these functions to compute x_s, y_s and z_e when X, Y and Z are given. The following functions perform this task:

```
inline double xScreen(int X) {return xsC + (X - XLC)/f;}
inline double yScreen(int Y) {return ysC + (Y - YLC)/f;}
inline double zEye(int Z) {return Z/zfactor + zemin;}
```

So much for the relationship between eye coordinates, screen coordinates and large coordinates. In the actual filling functions we have to convert the large coordinates X and Y to pixel coordinates, the range of which is dictated by our video display. The integers k and hk, computed above, are used for this in the following function:

```
inline int to_pix(int T) {return (T + hk)/k;}
```

This function is used both for X and Y, so we can write

```
Xpix = to_pix(X); Ypix = to_pix(Y);
```

The term hk is added in function **to_pix** to obtain the value of *Xpix* for which the absolute difference between $k \cdot Xpix$ and the given X is as small as possible. For example, if $k = 50$ and $X = 19990$, *Xpix* is computed as follows:

```
Xpix = (19990 + 25) / 50;
```

Since the operator /, applied to two integers, gives an integer result, this gives *Xpix* = 400, as if X had been 20000 instead of 19990.

7.3 A Simple Painter's Algorithm

It is an attractive idea to solve the hidden-face problem by displaying all polygons (which are triangles in our case) in a very specific order, namely first the most distant one, then the second farthest, and so on, finishing with the one that is closest to the viewpoint. Painters sometimes follow the same principle, starting with the background and painting a new layer for objects on the foreground later, so that some parts of objects in the background, painted previously, become invisible. This method is therefore known as the *painter's algorithm*. It is based on the assumption that each triangle can be assigned a z_e-coordinate, which we can use to sort all triangles. One way of obtaining such a z_e-coordinate for a triangle ABC is by computing the average of the three z_e-coordinates of A, B and C. However, there is a problem, which is clarified by Fig. 7.1. For each of the three beams, a large rectangle is partly obscured by another, so we cannot satisfactorily place them in the desired order, that is, from back to front. Consequently, the naive approach just suggested will fail in this case. Surprisingly enough, it gives good results in many other cases, and it is very fast. This

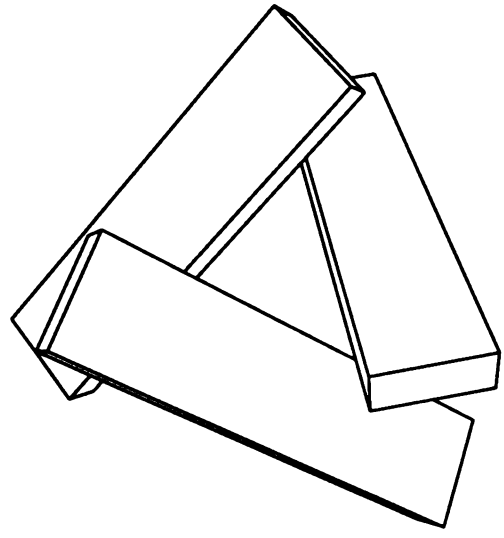


Fig. 7.1. Beams, each of which partly obscures another

is especially the case if, instead of floating-point coordinates z_e, we use large integer coordinates Z, as discussed in the previous section. We store such Z-values in the objects of type **tria**, the definition of which was given in Section 7.1.

Sorting can efficiently be done by using the **qsort** standard function, declared in the header file STDLIB.H. With the compare function

```
extern "C" int comparetriangles(const void *p1, const void *p2)
{  return ((tria*)p1)->Z < ((tria*)p2)->Z ? -1 : 1;
}
```

we can sort all triangles as follows:

```
qsort(triangles, ntr, sizeof(tria), comparetriangles);
```

(The prefix **extern "C"** as used above is required by some C++-compilers because **comparetriangles** is a C++ function which we supply as an argument to the C function **qsort**. Compilers that do not require this prefix accept it without any problems.) Since the triangle that is closest to the viewpoint is now in array element **triangles[0]** and the most distant one in **triangles[ntr-1]**, it is essential that i decreases in the following for-statement:

```
for (i=ntr-1; i>=0; i--) fill_triangle(i);
```

The simple painter's algorithm discussed here works correctly for many objects, including the torus shown in Fig. 8.9, and it is fast compared with other methods. It

is therefore included in program HIDEFACE, besides a more sophisticated algorithm, discussed in the next section. The user of the program can choose between these two alternatives.

7.4 Other Methods, Including Warnock's Algorithm

The problem with whole triangles (or other polygons), as used in the previous section, is that the relation 'obscures part of' is not transitive: if triangle T_1 obscures part of triangle T_2, which in turn obscures part of triangle T_3, then T_1 does not necessarily obscure part of T_3. What is worse, it is even possible for T_3 to obscure part of T_1, in the same way as is illustrated by Fig. 7.1 for three beams. Ordering a set of triangles according to increasing z_e-coordinate is dependent on our definition of 'the z_e-coordinate of a triangle'. By contrast, points on a line through viewpoint E can be ordered in a unique way. Let us use the term *ray* for such lines. It is possible to solve the hidden-face problem by considering a ray for each pixel of the screen. If a ray does not intersect any triangles, we display the pixel in the background color (or display nothing if the entire screen was filled with the background color beforehand). If it does intersect some triangles, there is a point of intersection of the ray for each triangle. We are then interested only in the one that is closest to E, and we display the pixel with the color belonging to the triangle in question. (Recall that the color to be used is derived from **dotproduct(n, l)**, where **n** is the normal vector of the triangle and **l** is the light vector.) With most computer systems (especially with high-resolution displays) this method will be slow because there are so may pixels to consider. While we were using only one z_e-coordinate for each triangle in the previous section, we are now using as many as there are pixels in the image of the triangle. It is an attractive idea to find some compromise between these two extremes, using sets of pixels lying, say, in small rectangles. The method to be discussed next is based on this idea.

Warnock's algorithm

An elegant way of dividing the screen into rectangles was presented by Warnock. We will again use the term *window* for rectangles (with horizontal and vertical sides) of the screen. Starting with a window which is the entire screen, windows are recursively divided into four smaller ones of equal size, as illustrated by Fig. 7.2. Each window consisting of more than one pixel is divided into four if it cannot be completely filled with one color.

Our implementation of Warnock's algorithm in program HIDEFACE (listed in Appendix B) is based on linked lists, each element of which contains only a triangle number (used as a subscript for array **triangles**) and a pointer to the next element, if any. Figure 7.3 shows the initial situation, with only one window, which is the whole screen. Associated with this window are the three triangles 0, 1, and 2; these numbers are stored in the linked list that belongs to this window.

If the situation is too complicated to be displayed at once, the window is divided into four, and a new linked list is constructed for each of these four smaller windows.

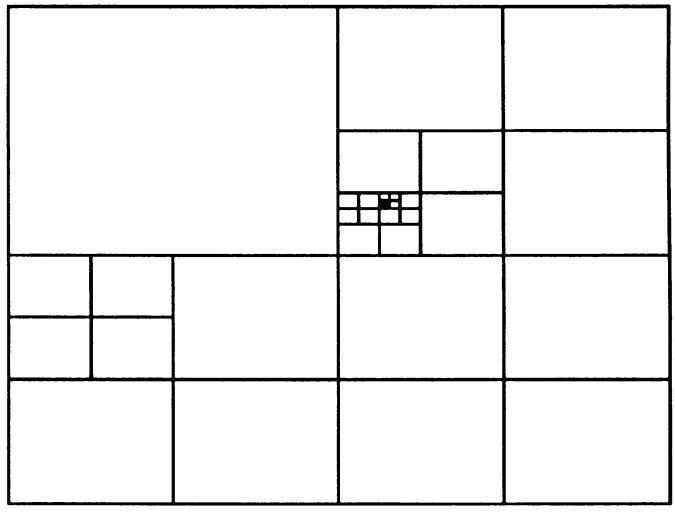


Fig. 7.2. Recursive division of windows

This is illustrated by Fig. 7.4; instead of one linked list of length 3, there are now three of length 2 and one of length 1. These four smaller windows are dealt with in the same way by means of recursive function calls.

Let us now discuss this method, implemented in program HIDEFACE (see Appendix B), in greater detail. The heart of this program is function **Warnock**, which has five parameters:

- *Xmin*, *Xmax*, *Ymin*, *Ymax*, of type **unsigned int**, to define the current window.
- *start*, the start pointer of the linked list containing all triangle numbers for this window.

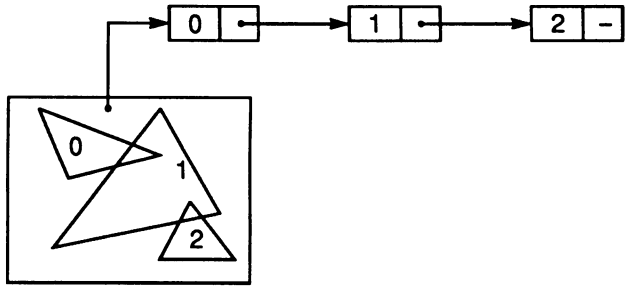


Fig. 7.3. The whole screen as initial window

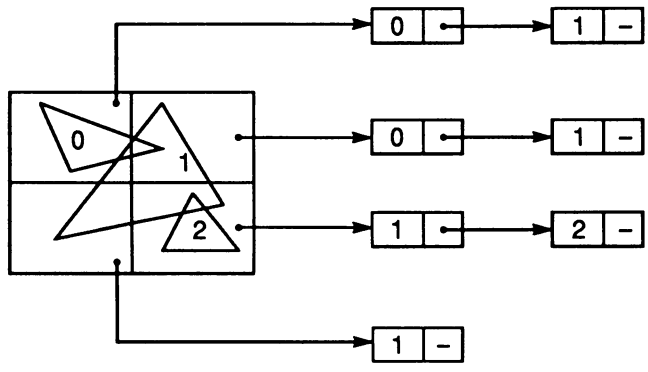


Fig. 7.4. Window divided into four

For each triangle given in the linked list, we test to see if its four corners lie inside it. If this is the case, we select the closest triangle, measured on a ray (that is, a line through viewpoint E) that passes through the center of the window. The color associated with the selected triangle is then used to fill the entire window. If the window does not surround all triangles given by the linked list, it is divided into four new ones, each with its own linked list of related triangles.

The rather time-consuming computation of each triangle's distance is required only to select the closest triangle, so it is omitted if only one triangle is related to the window.

This selection process must be carried out very carefully if the window consists of only one pixel. Suppose this pixel is point M(*Xmid*, *Ymid*) (expressed in 'large' coordinates, discussed in Section 7.2). Ray EM intersects the plane of a given triangle in point P. In this case, the window is represented only by point M; in other words, the four corners of the window, mentioned above, coincide. A single point is a dubious approximation of a window; selecting a triangle only on the basis of this single ray EM is therefore risky. With some objects, such as the spiral of beams discussed in Section 8.3 (see Plate 3 and Fig. 8.6), some pixels turned out to obtain a wrong color when only such a single ray was used. This can be explained as follows. Consider two beams, lying immediately above one another. They have two visible vertical faces, also one above the other, which lie in the same plane and have the same color. The edge shared by these two faces is also shared by the upper face of the lower beam. This horizontal face is obscured by the upper beam, but it is *not* a backface. If the ray we are using happens to end on the common edge just mentioned, endpoint P belongs to three instead of two triangles. Selecting the horizontal one instead of one of the two vertical ones would result in the wrong color.

The problem just mentioned is solved in program HIDEFACE by using also four other rays, which lie very close to ray EM. The width and height of the one-pixel window represented by point M are equal to k. It makes therefore sense to regard the four points (*Xmid*–*k*/2, *Ymid*–*k*/2), (*Xmid*+*k*/2, *Ymid*+*k*/2), (*Xmid*+*k*/2, *Ymid*–*k*/2) and

(*Xmid–k*/2, *Ymid+k*/2) as the corners of this tiny window. For each of these four new points, we consider a ray through it, which intersects the plane of a given triangle in point P′. If P′ lies inside this triangle (or on one of its edges), we use the length of EP′ as a distance, which will be slightly different from distance EP measured on ray EM. In this way we obtain at most five (but possibly fewer) distances, of which we compute the average. We select the triangle for which this average has the smallest value. So much for this rather technical detail, which applies only if more than one triangle is associated with a window consisting of only one pixel.

Program HIDEFACE, listed in Appendix B, contains many interesting geometric programming aspects, such as, for example, a test to see if a triangle has points in common with a window, as implemented in function **overlap**.

Remember, HIDEFACE accepts the same input files as HIDELINE, discussed in Chapter 6. Since Chapter 8 is about the generation of such files, its exercises also apply to our present subject and some of them are possibly more attractive to begin with than those listed below.

Exercises

7.1 If the object to be displayed is a convex polyhedron, we can simply fill each polygon that is not a backface (see also Exercise 6.1). Write a program to demonstrate this.

7.2 Find a way of extending program HIDEFACE so that 'loose' line segments (see Section 6.4) are also displayed, that is, as far as they are visible.

7.3 Program HIDEFACE is rather complex because of the use of integer coordinates. Replace this program with a simpler (but probably slower) version, based on floating-point coordinates.

7.4 Since we have assumed that our light source, like the sun, is practically infinitely far away, the same light vector was applied to all faces. This has the drawback that all parallel faces will have exactly the same color. Modify program HIDEFACE (or its simplified version, mentioned in Exercise 7.3) such that the position of a light source L instead of a light vector is read from the keyboard. Then each face can be assigned a light vector determined by L and the center of that face (or triangle).

8

Some Applications

8.1 Introduction

As we have seen, HIDELINE and HIDEFACE are general programs to display perspective images of solid objects bounded by flat faces. However, preparing data files for these programs involves a considerable amount of work. This task can be relieved in two ways. First, we can use existing programs to generate such files, and, second, we can write such programs ourselves. Since this book is about programming, we will focus on the latter solution. The programs in this chapter are therefore primarily intended as examples for those readers who want to write similar ones themselves. Those who do not want to program themselves but rather generate 3D data files by means of existing programs are also referred to *Interactive 3D Computer Graphics* (by the same author) and to the disks that accompany that book. Program D3D, discussed there, accepts the same files as the programs HIDELINE and HIDEFACE. The book mentioned offers a great many programs to generate such files. Besides, D3D itself can be used for this purpose: not only can it *read* such files, but it also offers good facilities for constructing 3D objects interactively and then *write* data files that describe them. These files can then be read later by program D3D itself but also by the programs HIDELINE and HIDEFACE. If these three programs are all available to you, you may think the situation rather confusing and wonder which to use. Summarizing their differences, we bear in mind that they all accept the same data files:

- D3D can be regarded as a '3D graphics editor': you can use it to construct 3D objects, read such objects from 3D data files, and write such files. It can display hidden-line representations of objects on request. Its use is limited to the IBM PC or compatible machines (with CGA, HGA, EGA or VGA).

- HIDELINE can only read 3D data files and display hidden-line representations. However, it does this much faster than D3D, and it allows many more vertices, lines and faces, that is, on the IBM PC. Although a ready-to-run version is available only for this machine (with VGA), machine-dependent aspects are isolated in a rather small module, GRSYS. It can therefore easily be made operational for other machines. It can also produce HP-GL files.
- HIDEFACE is essentially different from the other two programs in that it produces pictures with colored faces instead of line drawings. Such pictures look very realistic, as the color plates in this book illustrate. However, reproducing them is more costly. Machine-dependent aspects are restricted to module GRSYS.

As discussed in the Sections 6.2 and 6.4, the 3D data files discussed in this book have the following format:

vertex number x y z
⋮ ⋮ ⋮ ⋮
vertex number x y z

Faces:
vertex number ... *vertex number.*
⋮ ⋮
vertex number ... *vertex number.*

Remarks:

(1) The vertex numbers must be positive integers; the x-, y- and z-coordinates are real numbers, written in the usual way.

(2) The keyword **Faces** really is required; it must occur exactly once. It must be followed by sequences of vertex numbers. Each sequence must be followed by a period (.); the vertex numbers must be given in counter-clockwise order, when we are viewing the object from the outside. Vertices may be concave (in which case there will be three successive vertices that are clockwise); however, the first three vertices must be counter-clockwise, so the second vertex of the sequence must be convex.

(3) If a sequence consists of only two vertex numbers, it denotes that a simple line segment is to be drawn (as far as it is visible). This facility enables us to draw line segments that are not object edges. An exception is program HIDEFACE, which ignores such sequences.

(4) If a face has a hole, we transform it into a polygon by introducing an artificial edge. The latter will not be drawn if the second vertex number of the number pair for that edge is made negative. For example, if in Fig. 8.1 the inner rectangle is a hole, we can use the artificial edge (2, 6) as follows:

```
1  2  -6  5  8  7  6  -2  3  4.
```

If we follow the edges in this order, always facing the next vertex, we find the region we are describing on our left-hand side. This means that the vertices of the hole are traversed clockwise instead of counter-clockwise. The two minus signs in the above sequence indicate that (2, 6) and (6, 2) are not to be drawn. The first vertex number of a sequence must not be given a minus sign.

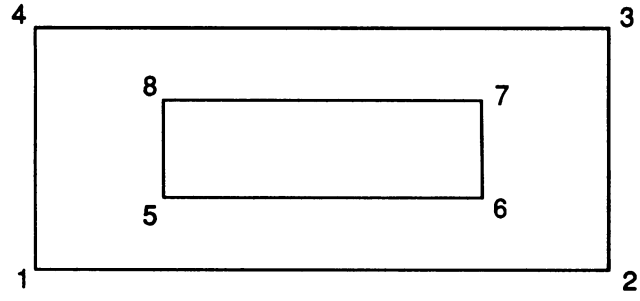


Fig. 8.1. Hole in rectangle

In this introduction we will use an object that is a logical extension of the example discussed in Section 6.4. Instead of a single solid letter A, we shall draw *n* copies of it in a row, where *n* can be any positive integer. In itself this picture is not likely to be of any practical value, but there is an aspect which we often encounter in practice, namely that a rather irregular portion of the picture (a single letter A) has a great many duplicates. Then the coordinates (and the vertex numbers) of the copies can be computed from original. The following program generates a 3D file of the above format and named LETTERSA.DAT. The program asks how many letters A are to be drawn:

```
// LETTERSA: Generating the 3D file LETTERSA.DAT
#include <fstream.h>
ofstream outf;

int f(int base, int x)
{  return x >= 0 ? x + base : x - base;
}

void writelist(int m, int base, int *a)
{  int i;
   for (i=0; i<m; i++) outf << " " << f(base, a[i]);
   outf << ".\n";
}
```

```
void rectangle(int base, int a, int b, int c, int d)
{  outf << f(base, a) << " " << f(base, b) << " "
       << f(base, c) << " " << f(base, d) << ".\n";
}

int main()
{  int thickness=10, n, i, j, nr, x, base;
   int X[21] = {0},
       Y[21] = {0, -30, -20, -16, 16, 20, 30,  0, -12, 12,  0},
       Z[21] = {0,   0,   0,   8,  8,  0,  0, 60,  16, 16, 40};
   cout << "How many letters A? "; cin >> n;
   outf.open("lettersa.dat");
   for (j=11; j<=20; j++)
   { X[j] = -thickness; Y[j] = Y[j-10]; Z[j]= Z[j-10];
   }
   for (i=0; i<n; i++)
   for (j=1; j<=20; j++)
   {  nr = 20*i+j; x = -2*i*thickness+X[j];
      outf << nr << " " << x << " " << Y[j] << " " << Z[i]
           << endl;
   }
   int front[12]={1, 2, 3, 4, 5, 6, 7, -10, 9, 8, 10, -7},
     back[12]={11, 17, -20, 18, 19, 20, -17, 16, 15, 14, 13, 12};
   outf << "Faces:\n";
   for (i=0; i<n; i++)
   {  writelist(12, base=20*i, front); writelist(12, base, back);
      rectangle(base, 2, 12, 13, 3);
      rectangle(base, 3, 13, 14, 4);
      rectangle(base, 15, 5, 4, 14);
      rectangle(base, 8, 9, 19, 18);
      rectangle(base, 8, 18, 20, 10);
      rectangle(base, 19, 9, 10, 20);
      rectangle(base, 6, 16, 17, 7);
      rectangle(base, 11, 1, 7, 17);
      rectangle(base, 11, 12, 2, 1);
      rectangle(base, 15, 16, 6, 5);
   }
   return 0;
}
```

Section 6.4 and, in particular, Fig. 6.16 will be helpful to understand this program. If we execute the program with 20 as the number of letters and 700, 80, 80 as the spherical coordinates of E, we obtain a quite extensive file LETTERSA.DAT. Executing HIDELINE with this file as input data gives the output of Fig. 8.2. The computing time on an IBM compatible 386 machine (20 MHz, no mathematical co-processor) was about 15s.

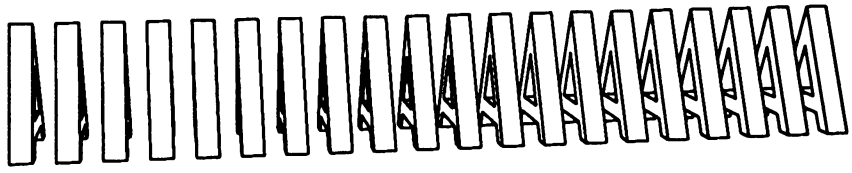


Fig. 8.2. Output produced by LETTERSA and HIDELINE

8.2 Hollow Cylinder

Many objects are bounded by curved surfaces. We can often approximate such surfaces by polygons. An example is a hollow cylinder as shown in Fig. 8.3(b). For some sufficiently large integer n, we choose n equidistant points on the outer circle (with radius R) of the top face, and we choose n similar points on the bottom face. Then we approximate the outer cylinder by a prism whose vertices are these $2n$ points. The inner circle (of the cilindrical hole) has radius r ($< R$). The hollow cylinder has height h. Let us use the z-axis of our coordinate system as the cylinder axis. The cylindrical hole is approximated by rectangles in the same way as the outer cylinder. The bottom face lies in the plane $z = 0$ and the top face in the plane $z = h$. A vertex of the bottom face lies on the positive x-axis. For given values n, R, r, h, the object to be drawn and its position are then completely determined. We shall first deal with the case $n = 6$ and generalize this later for arbitrary n. We number the vertices as shown in Fig. 8.4. For each vertex i of the top face ($1 \leq i \leq 12$) there is a vertical edge that connects it with vertex $i + 12$. We can specify the top face by means of the following sequence:

```
1   2   3   4   5   6 -12  11  10   9   8   7  12  -6.
```

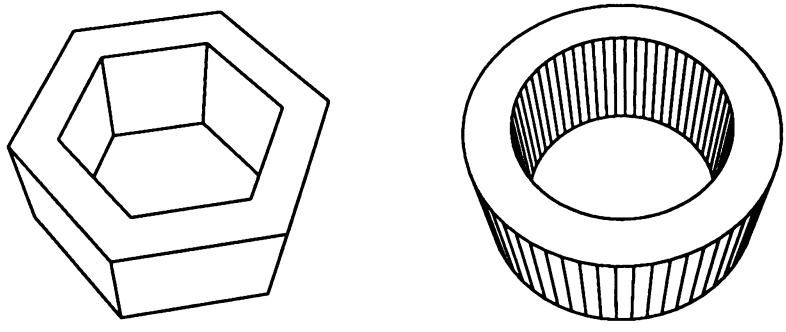


Fig. 8.3. (a) $n = 6$; (b) $n = 60$

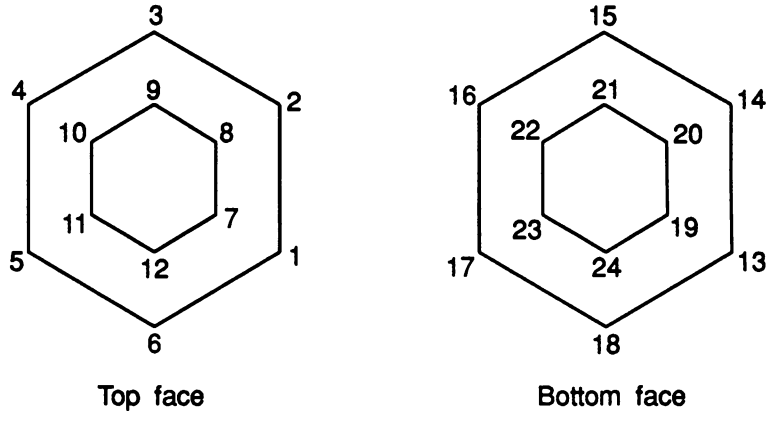


Fig. 8.4. Point numbering

Here the pairs (6, –12) and (12, –6) denote an artificial edge. In Fig. 8.4 the bottom face is viewed from the positive z-axis, but in reality only the other side is visible. The orientation of the bottom face is therefore opposite to what we see in Fig. 8.4 on the right, so that we can specify this face as

```
18 -24  19  20  21  22  23  24 -18  17  16  15  14  13.
```

Since $n = 6$, we have $12 = 2n$, $18 = 3n$ and $24 = 4n$, so the above sequences are special cases of

$$1 \ \ldots \ n \ -2n \ \ 2n-1 \ \ldots \ n+1 \ \ 2n \ -n.$$

and

$$3n \ -4n \ \ 3n+1 \ \ldots \ 4n \ -3n \ \ 3n-1 \ \ldots \ 2n+1.$$

Let us define

$$\delta = \frac{2\pi}{n}$$

Then the rectangular coordinates of the vertices on the top face (vertex numbers $i = 1,\ldots, 2n$) are

$$x_i = R \cos i\delta$$
$$y_i = R \sin i\delta \qquad (i = 1,\ldots, n; \text{ outer circle})$$
$$z_i = h$$

$$
\begin{aligned}
x_i &= r\cos(i-n)\delta \\
y_i &= r\sin(i-n)\delta \qquad (i = n+1,\ldots,2n;\ \text{inner circle}) \\
z_i &= h
\end{aligned}
$$

For the bottom face we have

$$
\begin{aligned}
x_i &= x_{i-2n} \\
y_i &= y_{i-2n} \qquad (i = 2n+1,\ldots,4n) \\
z_i &= 0
\end{aligned}
$$

All this is used in program HOLLOW, listed below. By choosing n large enough, we obtain a good approximation of a hollow cylinder, as Fig. 8.3(b) shows.

```
// HOLLOW: Generating a 3D file for a hollow cylinder
#include <fstream.h>
#include <math.h>

int main()
{  ofstream outf("hollow.dat");
   int n, j, k, l, i, m;
   float r, R, pi, alpha, cosa, sina, delta, h, radius, z;
   cout << "There are n points on a circle; n = ";
   cin >> n;
   cout << "Cylinder height = "; cin >> h;
   cout << "Large radius R and small radius r: ";
   cin >> R >> r;
   pi = 4.0 * atan(1.0); delta = 2.0 * pi/n;
   for (i=1; i<=n; i++)
   {  alpha = i * delta; cosa = cos(alpha); sina = sin(alpha);
      for (l=0; l<2; l++)     // l=0: outer,  l=1: inner circle
      {  radius= (l==0 ? R : r);
         for (m=0; m<2; m++) // m=0: top, m=1: bottom boundary
         {  k = i + l * n + 2 * m * n;
            z = (m == 0 ? h : 0);
            outf << k << " " << radius * cosa << " "
                 << radius * sina << " " << z << endl;
         }
      }
   }
   outf << "Faces:\n";
   // Top boundary face:
   for (i=1; i<=n; i++) outf << i << endl;
   outf << -2 * n << endl;
   for (i=2*n-1; i>=n+1; i--) outf << i << endl;
   outf << 2 * n << " " << -n << ".\n";
```

```
    // Bottom boundary face:
    outf << 3 * n << " " << -4 * n << endl;
    for (i=3*n+1; i<=4*n; i++) outf << i << endl;
    outf << -3 * n << endl;
    for (i=3*n-1; i>=2*n+2; i--) outf << i << endl;
    outf << (2 * n + 1) << ".\n";
    // Vertical lines:
    for (i=1; i<=n; i++)
    {  j = i % n + 1;
       outf << j << " " << i << " " << (i + 2 * n)
            << " " << (j + 2 * n) << ".\n";
       outf << (i + n) << " " << (j + n) << " " << (j + 3 * n)
            << " " << (i + 3 * n) << ".\n";
    }
    return 0;
}
```

8.3 Beams in a Spiral

Our next example is a spiral as shown in Fig. 8.6. It is build from horizontal beams with length l, width w and height w. The bottom beam lies in the xy-plane, as shown in Fig. 8.5. Starting at the bottom, each next beam position is obtained by rotating the preceding beam about the z-axis through 90° and by incrementing its z-coordinates by w at the same time. We number the beams 0, 1,..., n–1 from bottom to top. Beam i has vertex numbers $8i$+1, $8i$+2,..., $8i$+8, assigned in a systematic way, such that the vertex numbers of beam 0 are as shown in Fig. 8.5. Every point (x', y', z') of beam i + 1 can then be obtained by rotating the corresponding point (x, y, z) of beam i through 90° about the z-axis (and by setting $z' = z + w$). Thus we have

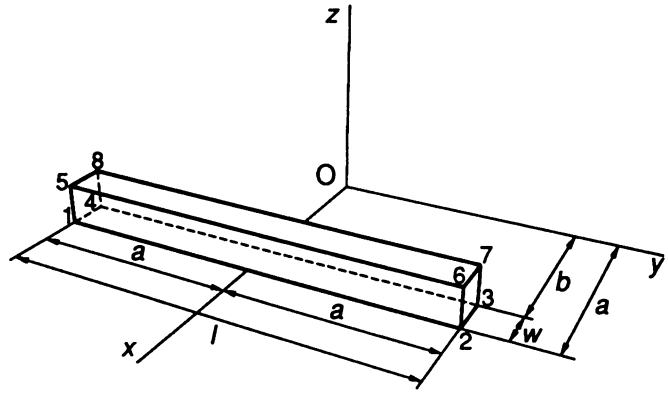


Fig. 8.5. Vertex numbers of beam 0

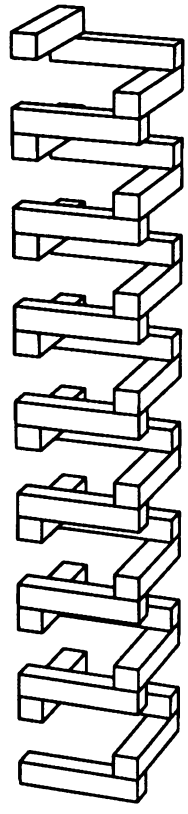


Fig. 8.6. Spiral of beams

$$[x' \;\; y'] = [x \;\; y] \begin{bmatrix} \cos 90° & \sin 90° \\ -\sin 90° & \cos 90° \end{bmatrix}$$

This can simply be written as $x' = -y$ and $y' = x$. All this is used in program BEAMS.

```
// BEAMS: Spiral of beams
#include <fstream.h>

int main()
{  ofstream outf("beams.dat");
   int i, j, n, A;
   float l, w, xA, yA, xB, yB, xC, yC, xD, yD, a, b, aux, z;
   cout << "How many beams? ";
```

```
    cin >> n;
    cout << "The beam measures l x w x w.\nEnter l and w:";
    cin >> l >> w;
    a = 0.5 * l;
    b = a - w;
    xA = a; yA = -a;
    xB = a; yB = a;
    xC = b; yC = a;
    xD = b; yD = -a;
    for (i=0; i<n; i++)
    {  for (j=0; j<2; j++)
       {  z = (i + j) * w;
          A = 8 * i + 4 * j;
          outf << (A+1) << " " << xA << " " << yA << " "
               << z << endl;
          outf << (A+2) << " " << xB << " " << yB << " "
               << z << endl;
          outf << (A+3) << " " << xC << " " << yC << " "
               << z << endl;
          outf << (A+4) << " " << xD << " " << yD << " "
               << z << endl;
       }
       aux = xA; xA = -yA; yA = aux;
       aux = xB; xB = -yB; yB = aux;
       aux = xC; xC = -yC; yC = aux;
       aux = xD; xD = -yD; yD = aux;
    }
    outf << "Faces:\n";
    for (i=0; i<n; i++)
    {  A = 8 * i;
       outf << (A+1) << " " << (A+4) << " "
            << (A+3) << " " << (A+2) << ".\n"; // Bottom
       outf << (A+5) << " " << (A+6) << " "
            << (A+7) << " " << (A+8) << ".\n"; // Top
       outf << (A+1) << " " << (A+2) << " "
            << (A+6) << " " << (A+5) << ".\n"; // Front
       outf << (A+4) << " " << (A+8) << " "
            << (A+7) << " " << (A+3) << ".\n"; // Back
       outf << (A+1) << " " << (A+5) << " "
            << (A+8) << " " << (A+4) << ".\n"; // Front
       outf << (A+2) << " " << (A+3) << " "
            << (A+7) << " " << (A+6) << ".\n"; // Back
    }
    return 0;
}
```

8.4 Spiral Staircase

The idea of beams in a spiral leads to our next example, a spiral staircase, as shown in Fig. 8.7. The vertical distance between two successive stairs is h. There are n stairs. Each has eight vertices, numbered 1,..., 8 for the stair at the bottom. We add the endpoints 9 and 10 of a vertical bar to them, which serves to attach a hand-rail to the stairs. The bars and the hand-rail are very thin, so we draw them as line segments. The center of the staircase is a cylindrical pole with diameter $2r$. Its axis coincides with the z-axis of our coordinate system. The hand-rail and the vertical bars are at a distance R from the z-axis ($R > r$). The stairs connect the pole with the vertical bars. The bottom stair is shown in Fig. 8.8. Each stair is a beam with length $R - r$, width $1.5h$ and height $0.2h$. Together the n stairs constitute a full revolution, that is, the angle of rotation for a single step is

$$\delta = \frac{2\pi}{n}$$

From bottom to top we assign the numbers 0, 1,..., $n - 1$ to the stairs. Stair i can be obtained by rotating stair 0 about the z-axis through the angle $\alpha = i\delta$ and by raising it to the height ih at the same time. Thus each vertex (x, y, z) of stair i can be computed from the corresponding vertex (X, Y, Z) of stair 0 as follows:

$$[x \quad y] = [X \quad Y] \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix}$$

$$z = Z + ih$$

In program STAIRS the vertex coordinates of the bottom stair are stored in the arrays X, Y, Z. We use the vertex numbers 1,..., $10n$ for the stairs, including the vertical bars and the hand-rail. After setting $M = 10n + 1$, we assign the numbers M,..., $M + n - 1$ to equidistant points on the bottom circle of the central pole. Finally, the integers $M + n$,..., $M + 2n - 1$ are assigned similarly to the top boundary of the pole.

```
// STAIRS: Spiral stairs
#include <fstream.h>
#include <math.h>

int main()
{  ofstream outf("stairs.dat");
   int i, j, n, k, M;
   float r, R, pi, alpha, cosa, sina, x, y, z,
         delta, h, H, H1, X[10], Y[10], Z[10];
   cout << "Enter n (to draw n stairs): ";
   cin >> n;
   cout << "Height of a single stair step: ";
   cin >> h;
```

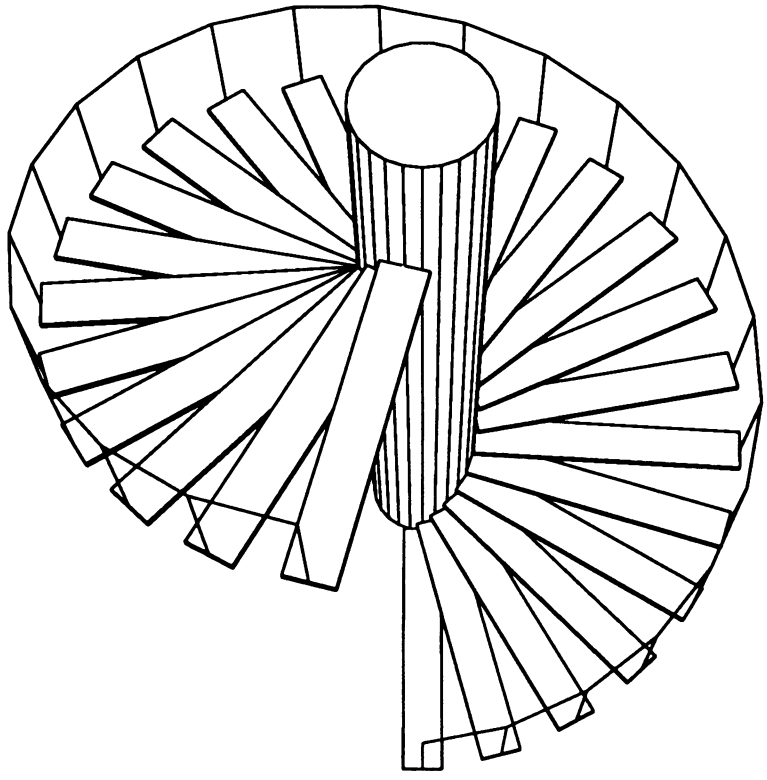


Fig. 8.7. Spiral staircase

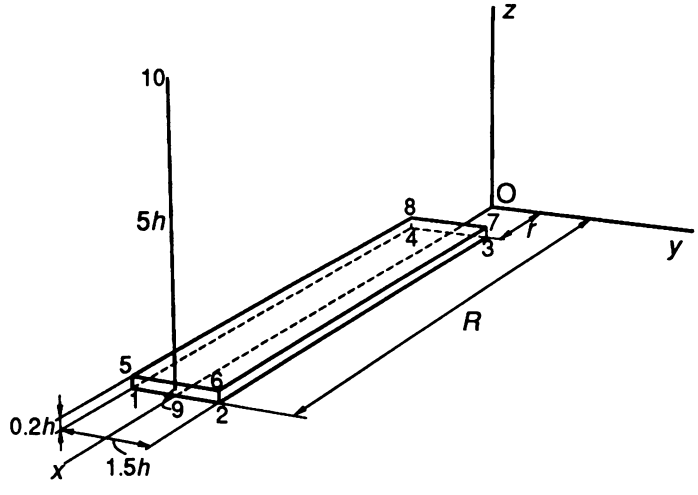


Fig. 8.8. Bottom stair

```
cout << "Large radius R and small radius r: ";
cin >> R >> r;
pi=4.0*atan(1.0); delta=2.0*pi/n;
H = n * h; H1 = H + 5 * h;
for (j=0; j<4; j++) Z[j]=0;
for (j=4; j<8; j++) Z[j]=h/5;
X[0]=X[1]=X[4]=X[5]=R;
X[2]=X[3]=X[6]=X[7]=r;
Y[0]=Y[4]=Y[3]=Y[7]=-.75*h;
Y[1]=Y[5]=Y[2]=Y[6]=.75*h;
X[8]=R; Y[8]=.0; Z[8]=h/10;
X[9]=R; Y[9]=.0; Z[9]=5*h;
M = 10 * n + 1; H = n * h;
for (i=0; i<n; i++)
{  alpha = i * delta;
   cosa = cos(alpha); sina = sin(alpha);
   for (j=0; j<10; j++)
   {  k = 10 * i + j + 1;
      x = X[j] * cosa - Y[j] * sina;
      y = X[j] * sina + Y[j] * cosa;
      z = Z[j] + i * h;
      outf << k << " " << x << " " << y <<
                  " " << z << endl;
   }
   x = r * cosa; y = r * sina;
   outf << M+i << " " << x << " " << y << " 0\n";
   outf << M+n+i << " " << x << " " << y << " " << H1 << endl;
}
outf << "Faces:\n";
for (i=0; i<n; i++)
{  k = 10 * i + 1;
   outf << k   << " " << k+1 << " " << k+5 << " "
        << k+4 << ".\n";
   outf << k+2 << " " << k+3 << " " << k+7 << " "
        << k+6 << ".\n";
   outf << k+1 << " " << k+2 << " " << k+6 << " "
        << k+5 << ".\n";
   outf << k+3 << " " << k << " "   << k+4 << " "
        << k+7 << ".\n";
   outf << k+2 << " " << k+1 << " " << k+5 << " "
        << k+6 << ".\n";
   outf << k+4 << " " << k+5 << " " << k+6 << " "
        << k+7 << ".\n";
   outf << k+1 << " " << k   << " " << k+3 << " "
        << k+2 << ".\n";
   outf << k+8 << " " << k+9 << ".\n";
```

```
        if (i < n-1) outf << k+9 << " " << k+19 << ".\n";
    }
    for (i=0; i<n; i++)
      outf << M+i << " " << M+(i+1)%n << " "
           << M+n+(i+1)%n << " " << M+n+i << ".\n";
    for (i=M+n-1; i>=M; i--) outf << " " << i;
    outf << ".\n";
    for (i=M+n; i<M+2*n; i++) outf << " " << i;
    outf << ".\n";
    return 0;
}
```

8.5 Torus

In the spiral staircase some dimensions (or rather proportions) were chosen arbitrarily. We will now discuss some examples where all vertices are computed from a very limited amount of data. The first is a torus, as shown in Fig. 8.9. The input of the program will consist of three numbers, n, R, r ($R > r$). In Fig. 8.10 the large horizontal circle is the set of centers of all small circles of the torus; the radius of this circle is R. We use n equidistant points on this circle as centers of small vertical circles with radius r. A parametric representation of the large circle is

$$x = R \cos \alpha$$
$$y = R \sin \alpha$$
$$z = 0$$

The point corresponding to $\alpha = 0$ is the center of the small circle with the following parametric representation:

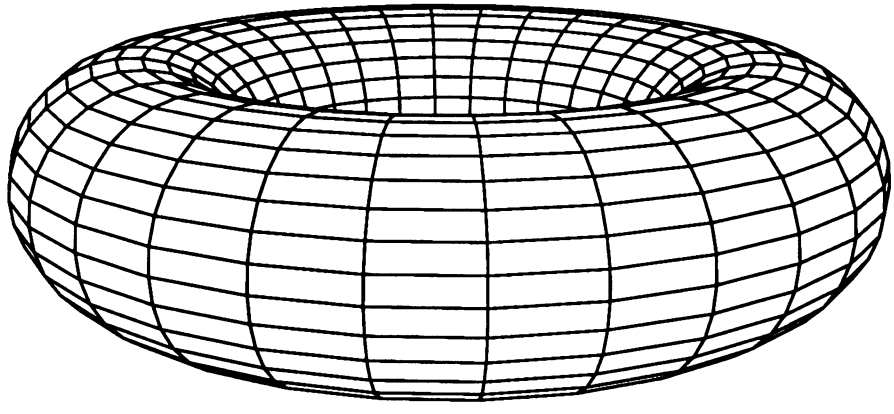


Fig. 8.9. Torus

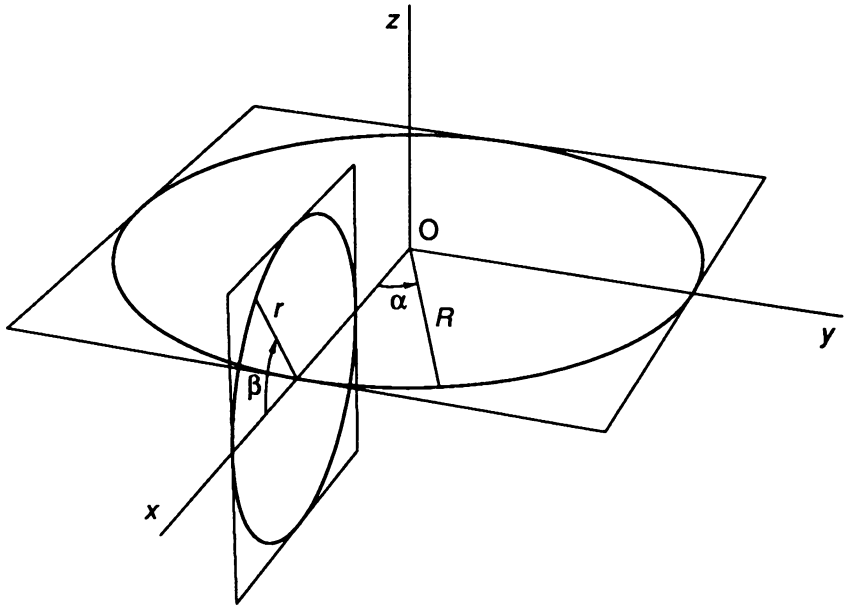


Fig. 8.10. Basic circles of torus

$$x = R + r \cos \beta$$
$$y = 0$$
$$z = r \sin \beta$$

This circle is also shown in Fig. 8.10. By rotating this particular small circle about the z-axis through angles $\alpha = i\delta$, where $i = 1,..., n - 1$ and $\delta = 2\pi/n$, we obtain the remaining $n - 1$ small circles. On the basic small circle in Fig. 8.10 we select n points and assign the vertex numbers $0, 1,..., n - 1$ to them: the point obtained by using parameter $\beta = j\delta$ is given vertex number j ($j = 0, 1,..., n - 1$). The next n vertices, numbered $n, n + 1,..., 2n - 1$, lie on the neighboring small circle, corresponding to $i = 1$, and so on. In general, we have the vertex numbers $i.n + j$ ($i = 0, 1,..., n - 1$; $j = 0, 1,..., n - 1$). As follows from Section 2.3, a rotation through the angle $\alpha = i\delta$ about the z-axis is written

$$[x' \quad y'] = [x \quad y] \begin{bmatrix} \cos \alpha & \sin \alpha \\ -\sin \alpha & \cos \alpha \end{bmatrix}$$

In our situation the basic small circle lies in the xz-plane, so $y = 0$, which reduces this matrix product to

$$x' = x \cos \alpha$$
$$y' = x \sin \alpha$$

This result could also have been derived immediately from Fig. 8.10. The following program generates the file for the torus.

```
// TORUS: Generating a 3D file for a torus
#include <fstream.h>
#include <math.h>
#include <stdlib.h>

int main()
{  ofstream outf("torus.dat");
   int i, j, n;
   float r, R, pi, alpha, beta, cosa, sina,
         x, x1, y1, z1, delta;
   cout << "Enter number n (to draw an n x n torus): ";
   cin >> n;
   cout << "Large radius R and small radius r: ";
   cin >> R >> r;
   pi = 4.0 * atan(1.0); delta = 2.0 * pi/n;
   for (i=0; i<n; i++)
   {  alpha=i*delta; cosa=cos(alpha); sina=sin(alpha);
      for (j=0; j<n; j++)
      {  beta=j*delta; x=R+r*cos(beta);          // y  = 0
         x1=cosa*x; y1=sina*x; z1=r*sin(beta);   // z1 = z
         outf << (i * n + j + 1) << " " << x1 << " " << y1
              << " " << z1 << endl;
      }
   }
   outf << "Faces:\n";
   for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      outf << (i * n + j + 1) << " " <<
              ((i + 1) % n * n + j + 1) << " " <<
              ((i + 1) % n * n + (j + 1) % n + 1) << " " <<
              (i * n + (j + 1) % n + 1) << ".\n";
   return 0;
}
```

8.6 Semi-sphere

Figure 8.11 shows the lower half of a sphere. We let the origin of the coordinate system coincide with the center of the sphere. Since the whole picture will automatically be as large as the viewport, the absolute size is irrelevant, so we can choose a radius of unit length. Thus all points (x, y, z) of the semi-sphere satisfy the following relations:

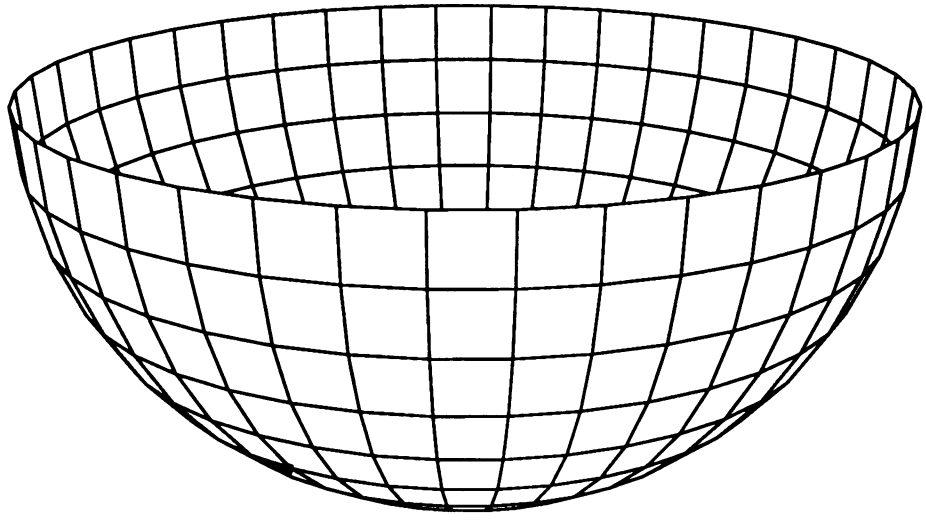


Fig. 8.11. Semi-sphere

$$\begin{cases} x^2 + y^2 + z^2 = 1 \\ -1 \le z \le 0 \end{cases}$$

We will divide each quadrant in the xy-plane into n equal angles $\delta = \frac{1}{2}\pi/n$; this number n is all our program will read. We will deal with the lowest point, the 'south pole', in a moment; let us first assign numbers to all other relevant points. Those with $x > 0$ and $y = 0$ are numbered 1, 2,..., n, counting from bottom to top. Their neighbors with $y > 0$ are numbered $n + 1$, $n + 2$,..., $2n$, and so on. This way of numbering is illustrated by the table below, where each (horizontal) row corresponds to points on a horizontal circle; similarly, each column corresponds to points on a quarter of a vertical circle. Since we define n^2 vertices for each of the four quadrants in this way, the first free number is $4n^2 + 1$, which is assigned to the 'south pole' (0, 0, –1).

j	$i = 0$	$i = 1$	$i = 2$		$i = 4n - 1$
n	n	2n	3n		$4n^2$
⋮	⋮	⋮	⋮	...	⋮
2	2	$n + 2$	$2n + 2$		$(4n - 1)n + 2$
1	1	$n + 1$	$2n + 1$		$(4n - 1)n + 1$

The $4n$ surface elements at the bottom deserve our special attention because they are triangles, each having the south pole as a vertex. The remaining $n - 1$ layers consist of quadrangles ABCD; each quadrangle has two edges AB and DC that are horizontal and parallel to each other. Note that this semi-sphere differs fundamentally from the other examples that we have seen. It is not a solid object but a surface, which is

visible from either side, depending on the viewpoint. Since each quadrangle has two faces, neither of them being a backface, we have to specify both sequences ABCD and DCBA in the following program, as discussed at the end of Section 6.4.

```
// SEMISPH: Generating a data file for a semi-sphere
#include <fstream.h>
#include <math.h>

int main()
{  ofstream outf("semisph.dat");
   int i, j, n, A, B, C, D, P, Q, southpole, n4;
   double pi, alpha, beta, delta, cosa, sina, cosb, sinb;
   cout << "There will be 4n points on each horizontal circle.\n";
   cout << "n = "; cin >> n;
   n4 = 4 * n;
   southpole = n4 * n + 1;
   pi = 4.0 * atan(1.0);
   delta = pi/(2 * n);
   // R = 1; sphere center in O
   outf << southpole << " 0 0 -1\n";
   for (i=0; i<n4; i++)
   {  alpha = i * delta;
      cosa = cos(alpha);
      sina = sin(alpha);
      for (j=1; j<=n; j++)
      {  beta = j * delta;
         cosb = cos(beta);
         sinb = sin(beta);
         outf << n*i+j << " " << sinb*cosa << " "
         << sinb*sina << " " << -cosb << endl;
      }
   }
   outf << "Faces:\n";
   for (i=0; i<n4; i++)
   {  P = i * n + 1;
      Q = (i < n4-1 ? P + n : 1);
      outf << southpole << " " << P << " " << Q << ".\n";
      outf << southpole << " " << Q << " " << P << ".\n";
      for (j=1; j<n; j++)
      {  A=P+j-1; B=Q+j-1; C=Q+j; D=P+j;
         outf << A << " " << B << " " << C << " " << D << ".\n";
         outf << D << " " << C << " " << B << " " << A << ".\n";
      }
   }
   return 0;
}
```

8.7 Functions of Two Variables

Program HIDELINE was primarily intended to represent solid objects. In two respects we can also use it for mathematical abstractions of such objects. First, 'loose' line segments, such as coordinate axes, can be drawn if we include their endpoints in the input file after the keyword 'Faces'. Second, objects may be surfaces, as is the case with the semi-sphere of the previous section. (As Plate 8 shows, images of surfaces can also be produced by program HIDEFACE.) This section is about another way of dealing with surfaces. We will consider functions of two variables:

$$z = f(x, y) \tag{8.1}$$

We specify a rectangular domain for this function as follows:

$$x_{\min} \leq x \leq x_{\max}$$
$$y_{\min} \leq y \leq y_{\max}$$

Although, in principle, function f can be any continuous function of two variables, let us use quadratic functions of the form

$$f(x, y) = ax^2 + by^2 + cxy + dx + ey + g \tag{8.2}$$

(We do not use the letter f as a coefficient, because f is the function name.) The program will ask for the coefficients a, b, c, d, e, g and also for the integers N_x and N_y. These two integers are used to compute the following length and width of elementary rectangles:

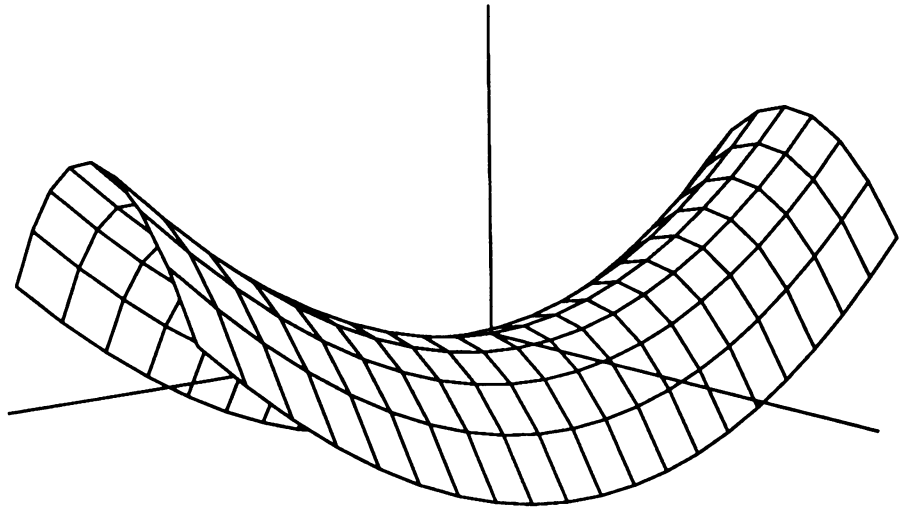


Fig. 8.12. A quadratic function of two variables

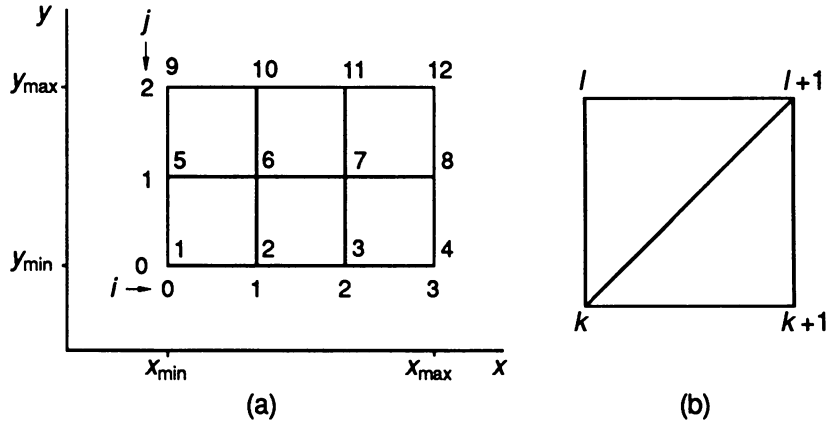


Fig. 8.13. (a) *Points on surface*; (b) *two triangles*

$$\Delta x = \frac{x_{max} - x_{min}}{N_x}$$

$$\Delta y = \frac{y_{max} - y_{min}}{N_y}$$

The corners of these rectangles are the grid points (x, y) for which we actually compute $z = f(x, y)$. The three-dimensional points (x, y, z) thus obtained are connected by straight line segments parallel to the xz- and the yz-planes. It is these line segments that will be drawn. Fig. 8.12 shows the function

$$f(x, y) = 0.1x^2 - 0.4y^2$$

where we have chosen

$-5 \le x \le 5 \quad -2 \le y \le 2$
$N_x = 20 \quad N_y = 8$
$\rho = 20 \quad \theta = 50° \quad \varphi = 80°$

We associate a pair of integers (i, j) with each grid point ($i = 0,..., N_x$, $j = 0,..., N_y$). Point (x, y, z), lying on the surface in question and corresponding to grid point (i, j), is assigned vertex number $k = j(N_x + 1) + i + 1$. Then we have

$$x = x_{min} + i \cdot \Delta x$$
$$y = y_{min} + i \cdot \Delta y$$
$$z = f(x, y)$$

The vertex numbering is illustrated in Fig. 8.13(a), where $N_x = 3$ and $N_y = 2$.

In Fig. 8.13(a), we view the surface from the positive z-axis. For example, points 1, 2, 5, 6 are points on the surface. Unfortunately, these four points will in general not lie in the same plane, so we cannot use them as vertices of a polygon. If we connect point 1 with point 6, we have two triangles that solve our problem. For the general case this is shown in Fig. 8.13(b), where $l = k + N_x + 1$. Though not co-planar, these two triangles can be used as the required polygons, provided that we prevent edge $(k, l + 1)$ from being drawn. Depending on our viewpoint, either side of the triangles can be visible, so under **Faces** we have to specify the triangles twice:

k	$-(l + 1)$	$k + 1.$	(lower-right, clockwise)
$k + 1$	$l + 1$	$-k.$	(lower-right, counter-clockwise)
k	$-(l + 1)$	$l.$	(upper-left, counter-clockwise)
l	$l + 1$	$-k.$	(upper-left, clockwise)

The minus signs will prevent line segments $(k, l + 1)$ from being drawn. Beside the function surface, we will also draw portions of the positive coordinate axes, as far as they are visible. Their lengths will be specified by the user.

For quadratic functions of two variables, the following program is quite general. For other functions $f(x, y)$, we can replace function f and remove the program elements that have to do with the coefficients a, b, c, d, e, and g.

```
/* FUNC: Generating a 3D file for a
         perspective plot of a quadratic function
*/
#include <fstream.h>

double a, b, c, d, e, g;

double f(double x, double y)
{  return a*x*x + b*y*y + c*x*y + d*x + e*y + g;
}

int main()
{  ofstream outf("func.dat");
   int i, j, Nx, Ny, k, l;
   double xmin, xmax, ymin, ymax, hx, hy, x, y,
          xaxis, yaxis, zaxis;
   cout << "f(x, y) = a.x.x + b.y.y + c.x.y + d.x + e.y + g\n";
   cout << "a, b, c, d, e, g: ";
   cin >> a >> b >> c >> d >> e >> g;
   cout << "xmin, xmax, ymin, ymax: ";
   cin >> xmin >> xmax >> ymin >> ymax;
   cout << "Grid parameters Nx, Ny: "; cin >> Nx >> Ny;
   hx = (xmax-xmin)/Nx; hy = (ymax-ymin)/Ny;
   cout << "Lengths of positive x-, y- and z-axes: ";
   cin >> xaxis >> yaxis >> zaxis;
```

```
    x = (xmin+xmax)/2; y = (ymin+ymax)/2;
    for (i=0; i<=Nx; i++)
    for (j=0; j<=Ny; j++)
    {  x = xmin + i * hx; y = ymin + j * hy;
       outf << j*(Nx+1)+i+1 << " " << x << " " << y
            << " " << f(x,y) << endl;
    }
    k = (Nx + 1) * (Ny + 1);
    outf << ++k << " 0 0 0\n";
    outf << ++k << " " << xaxis << " 0 0\n";
    outf << ++k << " 0 " << yaxis << " 0\n";
    outf << ++k << " 0 0 " << zaxis << "\n";
    outf << "Faces:\n";
    for (i=0; i<Nx; i++)
    for (j=0; j<Ny; j++)
    {  k = j * (Nx + 1) + i + 1; l = k + Nx + 1;
       outf << k << " " << -(l+1) << " " << k+1 << ".\n";
       outf << k+1 << " " << l+1 << " " << -k << ".\n";
       outf << k << " " << -(l+1) << " " << l << ".\n";
       outf << l << " " << l+1 << " " << -k << ".\n";
    }
    k = (Nx + 1) * (Ny + 1);
    outf << k+1 << " " << k+2 << ".\n";  // x-axis
    outf << k+1 << " " << k+3 << ".\n";  // y-axis
    outf << k+1 << " " << k+4 << ".\n";  // z-axis
    return 0;
}
```

Exercises

Like the programs listed in this chapter, the following exercises are about generating 3D files, which can be used as input for the programs HIDELINE and HIDEFACE.

8.1 Generate a 3D file for several pyramids, some of which partly hide others, depending on the viewpoint.

8.2 Generate a full sphere.

8.3 Generate several semi-spheres, some of which partly hide others.

8.4 Choose one of the Exercises of Chapter 5 and generate a 3D file for it. We are now in a position to give a solution that is more general than that in Chapter 5. Since hidden lines are now eliminated automatically, we can choose our viewpoint arbitrarily. Another advantage is that we can now also obtain color images by applying program HIDEFACE to the files that are generated.

8.5 Write a general rotation program that is based on module ROTA3D, described in Exercise 3.1. The user will specify a vector **AB** and an angle α. The program is to read a 3D file and to write another. In the latter file the coordinates will differ from those in the former, according to the given rotation, so that a picture of the rotated object will be the result.

8.6 Write a program that can combine two 3D files, resolving clashes of vertex numbers. The vertices of the second file are to be renumbered, so that in the combined file every vertex has its unique number.

8.7 Generate a great many cubes that are placed beside, behind and above each other (see Fig. 8.14).

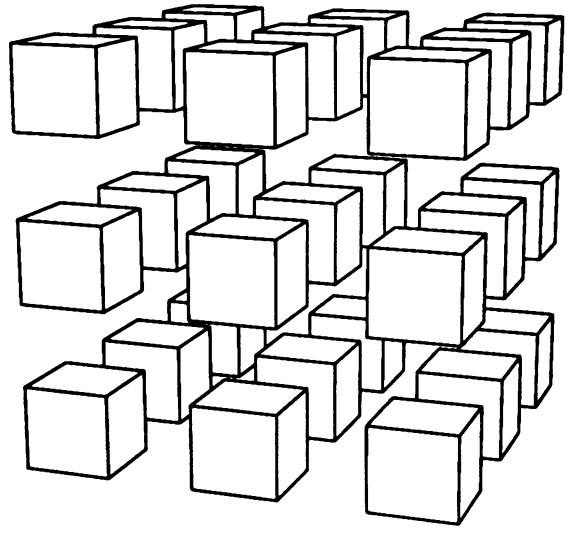


Fig. 8.14. A cube of cubes

8.8 Write a program that reads a 3D file and produces a similar one. The output file describes the same object as the input file, but shifted by a translation vector (Δx, Δy, Δz), read from the keyboard. Combining this with Exercises 8.5 and 8.6, and using a file produced by program TORUS (see Section 8.5), we can produce the three interlocking rings shown in Plate 5.

8.9 Write a program that combines Exercises 8.5, 8.6 and 8.8.

8.10 Write a program to generate to a solid object consisting of two cylinders in a T-form, as shown in Plate 7.

8.11 Write a program to generate a square screw thread, as shown in Fig. 8.15.

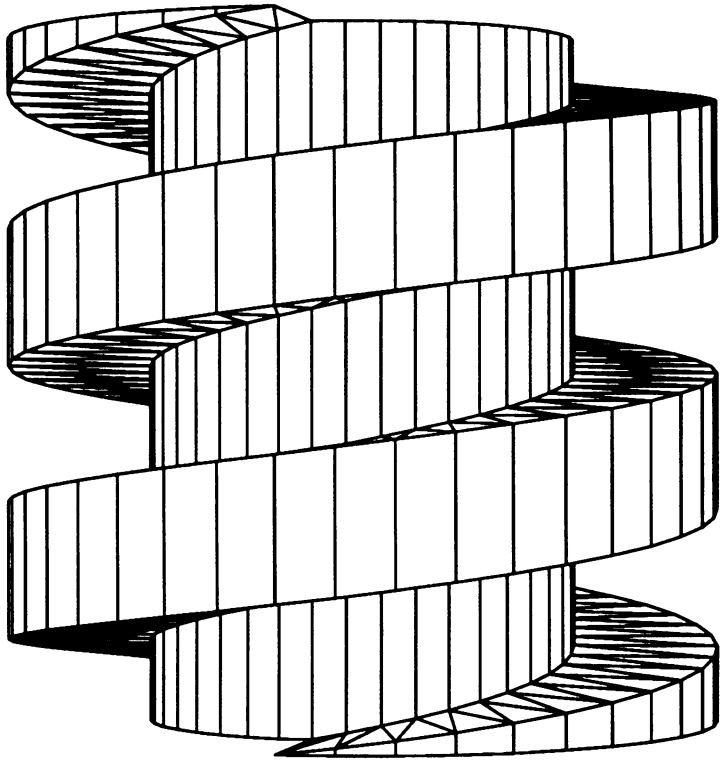


Fig. 8.15. Square screw thread

A

Program Text HIDELINE

```
/* HIDELINE: A program for hidden-line elimination.
      To be linked together with
         GRSYS (see Appendix C),
         D3 (see Section 5.3), and
         TRIANGUL (see Section 3.5).
*/
#include <fstream.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <time.h>
#include <limits.h>
#include "grsys.h"
#include "d3.h"
#include "triangul.h"

const int LARGE=32000,  // Not greater than sqrt(LONG_MAX)
          Nscreen=10;   // There will be Nscreen x Nscreen squares
const double PIdiv180=0.0174532, BIG=1.e30;
double d, c1, c2, xfactor, yfactor,
   Xrange, Yrange, Xvp_range, Yvp_range, Xmin, Xmax, Ymin,
   Ymax, deltaX, deltaY, denom, zemin=BIG, zemax=-BIG, eps1;
int *trset, dummy, vertexcount;

struct vec_int {int X; int Y;};
```

```
struct vertex{vec_int VT; double z; int *connect;} *V, *pvertex;
vec3 *vt;
struct triadata {vec3 normal; double h;};
struct tria {int Anr, Bnr, Cnr; triadata *ptria;}
   *triangles, *ptriangle;
struct node {int jtr; node *next;} *SCREEN[Nscreen][Nscreen];
struct point
{  vec_int Pntscr; double zPnt; int nrPnt;
   point(){}
   point(vec_int P, double z, int nr)
   {Pntscr = P; zPnt = z; nrPnt = nr;}
};

inline int max2(int x, int y){return x > y ? x : y;}
inline int min2(int x, int y){return x < y ? x : y;}
inline int max3(int x, int y, int z)
   {return x > y ? max2(x, z) : max2(y, z);}
inline int min3(int x, int y, int z)
   {return x < y ? min2(x, z) : min2(y, z);}
const int LARGE1 = LARGE + 1;
inline int colnr(int x) {return (long(x) * Nscreen) / LARGE1;}
inline int rownr(int y) {return (long(y) * Nscreen) / LARGE1;}
inline int Xcoord(int nr) {return (long(nr)*LARGE)/Nscreen;}
inline int xintscr(double x) {return (x - Xmin) * xfactor+.5;}
inline int yintscr(double y) {return (y - Ymin) * yfactor+.5;}
inline double xfloat(int x) {return x/xfactor + Xmin;}
inline double yfloat(int y) {return y/yfactor + Ymin;}

void memproblem() {errmess("Not enough memory");}

void skip(ifstream &inpfil) // Skip input characters until '\n'
{  char ch;
   inpfil.clear();
   do inpfil.get(ch); while (ch != '\n' && !inpfil.eof());
}

int orientv(vec_int &Ps, vec_int &Qs, vec_int &Rs)
{  int u1=Qs.X-Ps.X, u2=Qs.Y-Ps.Y,
       v1=Rs.X-Ps.X, v2=Rs.Y-Ps.Y;
   long det = long(u1) * v2 - long(u2) * v1;
   return det < -10 ? -1 : det > 10;
}

int orienta(int Pnr, int Qnr, int Rnr)
{  vec_int Ps=V[Pnr].VT, Qs=V[Qnr].VT, Rs=V[Rnr].VT;
   int u1=Qs.X-Ps.X, u2=Qs.Y-Ps.Y,
```

```
        v1=Rs.X-Ps.X, v2=Rs.Y-Ps.Y;
    long det = long(u1) * v2 - long(u2) * v1;
    return det < -300 ? -1 : det > 300;
}

struct linked_stack     // For deferred recursive calls
{   point P, Q;         // in function 'linesegment'
    int k0;
    linked_stack *next;
}   *stptr=NULL;

void stack_linesegment(point &P, point &Q, int k0)
{   linked_stack *p;
    int XP=P.Pntscr.X, YP=P.Pntscr.Y, XQ=Q.Pntscr.X, YQ=Q.Pntscr.Y;
    if (abs(XP-XQ) + abs(YP-YQ) < 50) return; // Not worthwhile
    p = stptr; stptr = new linked_stack;
    if (!stptr) memproblem();
    stptr->P = P; stptr->Q = Q; stptr->k0 = k0;
    stptr->next = p;
}

void linesegment(point &P, point &Q, int k0)
{   // Line segment PQ is to be drawn, as far as it is
    // not hidden by the triangles trset[0],..., trset[k0-1].
    vec_int Ps=P.Pntscr, Qs=Q.Pntscr, As, Bs, Cs, temp, Is, Js;
    double xP, yP, xQ, yQ, zP=P.zPnt, zQ=Q.zPnt, xI, yI,
       hP, hQ, xJ, yJ, lam_min, lam_max, lambda, mu,
       h, h1, h2, zI, zJ, zA, zB, zC, zmaxPQ;
    int Pnr=P.nrPnt, Qnr=Q.nrPnt,
       k=k0, j, Anr, Bnr, Cnr, i, Poutside, Qoutside,
       Pnear, Qnear,
       APB, AQB, BPC, BQC, CPA, CQA,
       xminPQ, xmaxPQ, yminPQ, ymaxPQ,
       XP=Ps.X, YP=Ps.Y, XQ=Qs.X, YQ=Qs.Y,
       u1=XQ-XP, u2=YQ-YP;
    long denom, v1, v2, w1, w2;
    vec3 normal;
    if (XP < XQ) {xminPQ = XP; xmaxPQ = XQ;}
            else {xminPQ = XQ; xmaxPQ = XP;}
    if (YP < YQ) {yminPQ = YP; ymaxPQ = YQ;}
            else {yminPQ = YQ; ymaxPQ = YP;}
    while (k > 0)
    {   j = trset[--k];
       Anr = triangles[j].Anr;
       Bnr = triangles[j].Bnr;
       Cnr = triangles[j].Cnr;
```

```
// Test 1 (3D): Is PQ one of the triangle edges?
if ((Pnr == Anr || Pnr == Bnr || Pnr == Cnr) &&
    (Qnr == Anr || Qnr == Bnr || Qnr == Cnr)) continue;
As = V[Anr].VT; Bs = V[Bnr].VT; Cs = V[Cnr].VT;

// Test 2 (2D): Minimax tests:
if (xmaxPQ <= As.X && xmaxPQ <= Bs.X && xmaxPQ <= Cs.X ||
    xminPQ >= As.X && xminPQ >= Bs.X && xminPQ >= Cs.X ||
    ymaxPQ <= As.Y && ymaxPQ <= Bs.Y && ymaxPQ <= Cs.Y ||
    yminPQ >= As.Y && yminPQ >= Bs.Y && yminPQ >= Cs.Y)
        continue; // continue means: 'visible'

// Test 3 (2D): Do P and Q lie in a half plane defined by
//     an edge of triangle ABC (and outside this triangle)?
APB = orientv(As, Ps, Bs); AQB = orientv(As, Qs, Bs);
if (APB + AQB > 0) continue;
BPC = orientv(Bs, Ps, Cs); BQC = orientv(Bs, Qs, Cs);
if (BPC + BQC > 0) continue;
CPA = orientv(Cs, Ps, As); CQA = orientv(Cs, Qs, As);
if (CPA + CQA > 0) continue;

// Test 4 (2D): Do A, B and C lie in the same half plane
//              defined by PQ?:
if (abs(orientv(Ps, Qs, As) + orientv(Ps, Qs, Bs) +
        orientv(Ps, Qs, Cs)) > 1) continue;

// Test 5 (3D): Are both zP and zQ less than zA, zB and zC?
zA = V[Anr].z; zB = V[Bnr].z; zC = V[Cnr].z;
zmaxPQ = (zP > zQ ? zP : zQ);
if (zmaxPQ <= zA && zmaxPQ <= zB && zmaxPQ <= zC) continue;

// Test 6 (3D): Does neither P nor Q lie behind plane ABC?
normal = triangles[j].ptria->normal;
h = triangles[j].ptria->h;
if (h == 0) continue; // Plane through viewpoint
xP = zP * xfloat(XP); yP = zP * yfloat(YP);
xQ = zQ * xfloat(XQ); yQ = zQ * yfloat(YQ);
hP = normal.x * xP + normal.y * yP + normal.z * zP;
hQ = normal.x * xQ + normal.y * yQ + normal.z * zQ;
h2 = h + eps1;
if (hP <= h2 && hQ <= h2) continue;

// Test 7 (2D) Does triangle ABC completely obscure PQ?
Poutside = APB == 1 || BPC == 1 || CPA == 1;
Qoutside = AQB == 1 || BQC == 1 || CQA == 1;
if (!Poutside && !Qoutside) return;
```

```
// None of the preceding continue-statements were executed,
// so line segment PsQs has points in common with triangle
//  AsBsCs.
h1 = h - eps1;
Pnear = hP < h1; Qnear = hQ < h1;
if (Pnear && !Poutside || Qnear && !Qoutside) continue;
// Now P lies either outside pyramid EABC or behind
// triangle ABC, and the same applies to Q.

// The points of intersection are now computed:
lam_min = 1.0; lam_max = 0.0;
for (i=0; i<3; i++)
{  v1 = Bs.X - As.X; v2 = Bs.Y - As.Y;
   w1 = As.X - XP; w2 = As.Y - YP;
   denom = u1 * v2 - u2 * v1;
   if (denom != 0) // PsQs not parallel to AsBs
   {  mu = (u2 * w1 - u1 * w2)/double(denom);
      // mu = 0 gives A, and mu = 1 gives B.
      if (mu > -0.0001 && mu < 1.0001)
      {  lambda = (v2 * w1 - v1 * w2)/double(denom);
         // lambda = PI/PQ (I is point of intersection)
         if (lambda > -.0001 && lambda < 1.0001)
         {  if (Poutside != Qoutside &&
            lambda > .0001 && lambda < .9999)
            {  lam_min = lam_max = lambda;
               break;  // Only one point of intersection
            }
            if (lambda < lam_min) lam_min = lambda;
            if (lambda > lam_max) lam_max = lambda;
         }
      }
   }
   temp = As; As = Bs; Bs = Cs; Cs = temp;
}
//  Test 8: I and J are points of intersection.
//  Test whether these points lie in front of triangle ABC:
if (Poutside && lam_min > .01)
{  Is.X = int(XP + lam_min * u1 + 0.5);
   Is.Y = int(YP + lam_min * u2 + 0.5);
   zI = 1/(lam_min/zQ + (1 - lam_min)/zP);
   xI = zI * xfloat(Is.X); yI = zI * yfloat(Is.Y);
   if (normal.x * xI + normal.y * yI + normal.z * zI < h1)
      continue;
   stack_linesegment(point(Ps, zP, Pnr),
                     point(Is, zI, -1), k);
}
```

```
      if (Qoutside && lam_max < .99)
      {  Js.X = int(XP + lam_max * u1 + 0.5);
         Js.Y = int(YP + lam_max * u2 + 0.5);
         zJ = 1/(lam_max/zQ + (1 - lam_max)/zP);
         xJ = zJ * xfloat(Js.X); yJ = zJ * yfloat(Js.Y);
         if (normal.x * xJ + normal.y * yJ + normal.z * zJ < h1)
            continue;
         stack_linesegment(point(Qs, zQ, Qnr),
                           point(Js, zJ, -1), k);
      }
      return; // Only if no continue-statement has been executed
   }
   move(d * xfloat(XP) + c1, d * yfloat(YP) + c2);
   draw(d * xfloat(XQ) + c1, d * yfloat(YQ) + c2);
}

void dealwithlinkedstack()
{  linked_stack *p;        // Deferred recursive calls
   point P, Q;             // stored in a linked stack
   int k0;
   while (stptr)
   {  p = stptr; P = p->P; Q = p->Q; k0 = p->k0;
      stptr = p->next;
      delete p;
      linesegment(P, Q, k0);
   }
}

void add_linesegment(int Pnr, int Qnr)
{  int iaux, *p, *p_old, i, n;
   if (Pnr > Qnr) {iaux = Pnr; Pnr = Qnr; Qnr = iaux;}
   // Now: Pnr < Qnr
   p = V[Pnr].connect;
   if (p == NULL)
   {  V[Pnr].connect = p = new int[3];
      if (p == NULL) memproblem();
      *p = 1; p[1] = Qnr; // *p=n=1, p[1]=Qnr (p[2] free)
      return;
   }
   n = *p;
   for (i=1; i<=n; i++) if (p[i] == Qnr) return; // Already in list
   n++; // Q is to be placed into p[n]
   if (n % 3 == 0)
   {  p_old = p;
      V[Pnr].connect = p = new int[n+3]; // Blocks of 3 integers
      if (p == NULL) memproblem();
```

```
      for (i=1; i<n; i++) p[i] = p_old[i];
      *p = n; p[n] = Qnr;  // n is a multiple of 3
      delete p_old;  // *p=n, p[1],..., p[n] used
                     // (p[n+1], p[n+2] free)
   }  else
   {  *p = n; p[n] = Qnr; // n not a multiple of 3 (and n > 1)
   }
}

void setupscreenlists(tria *TR, int n)
{  // Set up screen lists of triangles given as TR[0],..., TR[n-1].
   int i, l, I, J, Imin, Imax, j_old, jI, topcode[3],
      ileft, iright, LOWER[Nscreen], UPPER[Nscreen];
   long deltax, deltay;
   vec_int As, Bs, Cs, Left[3], Right[3], Aux;
   tria *p;
   node *p_new, *p_old;
   for (i=0; i<n; i++)
   {  p = TR + i;
      As = V[p->Anr].VT; Bs = V[p->Bnr].VT; Cs = V[p->Cnr].VT;
      topcode[0] = As.X > Bs.X; // Because of positive orientation
      topcode[1] = Cs.X > As.X;
      topcode[2] = Bs.X > Cs.X;
      Left[0] = As; Right[0] = Bs;
      Left[1] = As; Right[1] = Cs;
      Left[2] = Bs; Right[2] = Cs;
      for (l=0; l<3; l++)  // l = triangle-side number
         if (Left[l].X > Right[l].X ||
         (Left[l].X == Right[l].X && Left[l].Y > Right[l].Y))
         {  Aux = Left[l]; Left[l] = Right[l]; Right[l] = Aux;
         }
      Imin = colnr(min3(As.X, Bs.X, Cs.X));
      Imax = colnr(max3(As.X, Bs.X, Cs.X));
      for (I = Imin; I<=Imax; I++)
      {  LOWER[I] = INT_MAX; UPPER[I] = INT_MIN;
      }
      for (l=0; l<3; l++)
      {  ileft = colnr(Left[l].X); iright = colnr(Right[l].X);
         if (ileft != iright)
         { deltay = Right[l].Y - Left[l].Y;
           deltax = Right[l].X - Left[l].X;
         }
         j_old = rownr(Left[l].Y);
         for (I=ileft; I<=iright; I++)
         {  jI = (I == iright ? rownr(Right[l].Y) : rownr(Left[l].Y
                  + (Xcoord(I+1) - Left[l].X) * deltay / deltax));
```

```
            if (topcode[l])
                UPPER[I] = max3(j_old, jI, UPPER[I]);
            else LOWER[I] = min3(j_old, jI, LOWER[I]);
            j_old = jI;
         }
      }
      // For screen column I, the triangle is associated only
      // with rectangles in the rows LOWER[I],...,UPPER[I].
      for (I=Imin; I<=Imax; I++)
      for (J=LOWER[I]; J<=UPPER[I]; J++)
      {  p_old = SCREEN[I][J];
         SCREEN[I][J] = p_new = new node;
         if (p_new == NULL) memproblem();
         p_new->jtr = i; p_new->next = p_old;
      }
   }
}

void complete_triangles(int n, int offset, trianrs *nrs_tr)
// Complete triangles[offset],..., triangles[offset+n-1].
// Vertex numbers: nrs_tr[0],..., nrs_tr[n-1].
// These triangles belong to the same polygon. The equation
// of their plane is nx . x + ny . y + nz . z = h.
{  int i, Anr, Bnr, Cnr;
   unsigned Zmin, Zmax;
   double nx, ny, nz, ux, uy, uz, vx, vy, vz, factor, h,
      Ax, Ay, Az, Bx, By, Bz, Cx, Cy, Cz;
   tria *p;
   triadata *q = new triadata;
   if (q == NULL) memproblem();
   // If the polygon is an approximation of a circle, the
   // first three vertices my lie almost on the same line,
   // hence n/2 instead of 0 in the following for-statement:
   for (i = n/2; i<n; i++)
   {  Anr = nrs_tr[i].A;
      Bnr = nrs_tr[i].B;
      Cnr = nrs_tr[i].C;
      if (orienta(Anr, Bnr, Cnr) > 0) break;
   }
   Az = V[Anr].z; Bz = V[Bnr].z; Cz = V[Cnr].z;
   Ax = xfloat(V[Anr].VT.X) * Az; Ay = yfloat(V[Anr].VT.Y) * Az;
   Bx = xfloat(V[Bnr].VT.X) * Bz; By = yfloat(V[Bnr].VT.Y) * Bz;
   Cx = xfloat(V[Cnr].VT.X) * Cz; Cy = yfloat(V[Cnr].VT.Y) * Cz;
   // Computation of vector product normvec = (B-A) x (C-A):
   ux = Bx - Ax; uy = By - Ay; uz = Bz - Az;
   vx = Cx - Ax; vy = Cy - Ay; vz = Cz - Az;
```

```
   nx = uy * vz - uz * vy;
   ny = uz * vx - ux * vz;
   nz = ux * vy - uy * vx;
   h = nx * Ax + ny * Ay + nz * Az;
   factor = 1/sqrt(nx * nx + ny * ny + nz * nz);
   q->normal.x = nx * factor;
   q->normal.y = ny * factor;
   q->normal.z = nz * factor;
   q->h = h * factor;
   for (i=0; i<n; i++)
   {  p = triangles + offset + i;
      p->Anr = nrs_tr[i].A;
      p->Bnr = nrs_tr[i].B;
      p->Cnr = nrs_tr[i].C;
      p->ptria = q;
   }
}

int main(int argc, char *argv[])
{  int i, Pnr, Qnr, ii, vertexnr, minvertnr=INT_MAX, maxvertnr=0,
      *ptr, iconnect, code, ntr=0, I, J, jtop, jbot, jI,
      trnr, jtr, *POLY, npoly, ileft, iright, nvertex, ntrset,
      maxntrset=400, LOWER[Nscreen], UPPER[Nscreen], orient,
      maxnpoly, totnrtria, testtria[3];
   clock_t t1, t2;
   trianrs *nrs_tr;
   char ch;
   long deltax, deltay;
   double rho, theta, phi, X, Y, xe, ye, ze, x, y, z,
      fx, fy, Xcenter, Ycenter, xmin=BIG, xmax=-BIG,
      ymin=BIG, ymax=-BIG, zmin=BIG, zmax=-BIG;
   vec_int Ps, Qs, Left, Right;
   ifstream inpfil;
   vec3 ve, P, Objpoint, Pnew;
   node *pnode;
   if (argc<2 || (inpfil.open(argv[1]), !inpfil))
      errmess("Input file not correctly specified.");
   // Initialize screen matrix:
   for (I=0; I<Nscreen; I++)
   for (J=0; J<Nscreen; J++) SCREEN[I][J] = NULL;
   cout << "Please wait...\n";
   for ( ; ; )
   {  inpfil >> i >> x >> y >> z;
      if (i % 50 == 0) cout << ".";
      if (i < minvertnr) minvertnr = i;
      if (i > maxvertnr) maxvertnr = i;
```

```
   if (x < xmin) xmin = x; if (x > xmax) xmax = x;
   if (y < ymin) ymin = y; if (y > ymax) ymax = y;
   if (z < zmin) zmin = z; if (z > zmax) zmax = z;
   if (inpfil.fail() || inpfil.eof()) break;
}
cout << endl;
inpfil.close(); inpfil.open(argv[1]);
nvertex = maxvertnr + 1;
vt = new vec3[nvertex];  // Preliminary vertex list
if (!vt) errmess("Too many vertices");
Objpoint =
   vec3(.5*(xmin+xmax), .5*(ymin+ymax), .5*(zmin+zmax));
cout << "x range: " << xmin << " ... " << xmax << endl;
cout << "y range: " << ymin << " ... " << ymax << endl;
cout << "z range: " << zmin << " ... " << zmax << endl;
cout << "Center: " << Objpoint.x << " " << Objpoint.y << " "
     << Objpoint.z << endl;
cout << "Use center as default central object point? (Y/N): ";
cin >> ch; ch = toupper(ch);
if (ch == 'N')
{  cout << "Enter coordinates of central object point:\n";
   cin >> Objpoint;
}
rho = xmax - xmin;
if (ymax - ymin > rho) rho = ymax - ymin;
if (zmax - zmin > rho) rho = zmax - zmin;
rho *= 3; theta = 20; phi = 70;
cout << "Use default viewpoint (rho = " << rho
     << ", theta = 20, phi = 70)? (Y/N): ";
cin >> ch; ch = toupper(ch);
if (ch == 'N')
{  cout << "Enter spherical coordinates rho, theta, phi of\n";
   cout << "viewpoint E (phi = angle between z-axis and OE)\n";
   cin >> rho >> theta >> phi;
}
t1 = clock();
coeff(rho, theta*PIdiv180, phi*PIdiv180);
for (i=0; i<nvertex; i++) vt[i].z = -1e6; // Code for not in use
// Read vertices:
Xmin=Ymin=BIG; Xmax=Ymax=-BIG;
cout << "Please wait...\n";
for ( ; ; )
{  inpfil >> i >> P;
   if (i % 50 == 0)
      cout << "."; // Show that something is happening...
   if (inpfil.fail() || inpfil.eof()) break;
```

```
      vertexcount++;
      if (i < 0 || i >= nvertex)
         errmess("Too many vertices or illegal vertex number");
      Pnew.x = P.x - Objpoint.x;
      Pnew.y = P.y - Objpoint.y;
      Pnew.z = P.z - Objpoint.z;
      eyecoord(Pnew, ve);
      xe = ve.x; ye = ve.y; ze = ve.z;
      if (ze < 0)
         errmess("\nObject point O and a vertex on different "
                 "sides of viewpoint E.\n"
                 "Try a larger value for rho.\n");
      if (ze < zemin) zemin = ze;
      if (ze > zemax) zemax = ze;
      X = xe/ze;  Y = ye/ze;
      if (X < Xmin) Xmin = X; if (X > Xmax) Xmax = X;
      if (Y < Ymin) Ymin = Y; if (Y > Ymax) Ymax = Y;
      vt[i] = ve;
   }
   cout << endl;
   eps1 = .001 * (zemax - zemin);
   if (Xmin == BIG) errmess("Incorrect input file");
   if (argc == 2) initgr(); else initgr(argv[2]);
   // Compute screen constants:
   Xrange = Xmax - Xmin; Yrange = Ymax - Ymin;
   Xvp_range = x_max - x_min; Yvp_range = y_max - y_min;
   fx = Xvp_range/Xrange; fy = Yvp_range/Yrange;
   d = (fx < fy ? fx : fy);
   Xcenter = 0.5 * (Xmin + Xmax); Ycenter = 0.5 * (Ymin + Ymax);
   c1 = x_center - d * Xcenter; c2 = y_center - d * Ycenter;
   deltaX = Xrange/Nscreen; deltaY = Yrange/Nscreen;
   xfactor = LARGE/Xrange; yfactor = LARGE/Yrange;
   V = new vertex[nvertex]; if (V == NULL) memproblem();
   // Initialize vertex array:
   for (i=0; i<nvertex; i++)
   {  if (vt[i].z < -1e5) {V[i].connect = &dummy; continue;}
                          // V[i] not in use.
      V[i].connect = NULL;
      V[i].VT.X = xintscr(vt[i].x / vt[i].z);
      V[i].VT.Y = yintscr(vt[i].y / vt[i].z);
      V[i].z = vt[i].z;
   }
   delete[] vt;  // Replaced with V
   // Find the maximum number of vertices in one polygon
   // and the total number of non-backface triangles:
   skip(inpfil); // Skip the word 'Faces'
```

```
maxnpoly = totnrtria = 0;
for ( ; ; )
{  for (npoly=0; ; npoly++)
   {  inpfil >> i; i = abs(i);
      if (inpfil.fail() || inpfil.eof()) break;
      if (i >= nvertex || V[i].connect == &dummy)
         errmess("Undefined vertex number used.");
      if (npoly < 3) testtria[npoly] = i;
   }
   if (inpfil.eof()) break;
   if (npoly > maxnpoly) maxnpoly = npoly;
   skip(inpfil);
   if (npoly < 3) continue; // Ignore 'loose' line segment
   if (orienta(testtria[0], testtria[1], testtria[2]) >= 0)
   totnrtria += npoly - 2;
}
inpfil.close(); inpfil.open(argv[1]); // Rewind
do inpfil >> x; while (!inpfil.fail() && !inpfil.eof());
triangles = new tria[totnrtria];
POLY = new int[maxnpoly];
nrs_tr = new trianrs[maxnpoly-2];
if (!triangles || !POLY || !nrs_tr) memproblem();

// Read object faces and store triangles:
skip(inpfil); // To skip the word 'Faces'
for ( ; ; )
{  npoly = 0;
   for ( ; ; )
   {  inpfil >> i;
      if (inpfil.fail() || inpfil.eof()) break;
      POLY[npoly] = i;
      i = abs(i);
      if (i >= nvertex || V[i].connect == &dummy)
         errmess("Undefined vertex number used.");
      npoly++;
   }
   if (inpfil.eof()) break;
   skip(inpfil);
   if (npoly==1) errmess("Only one vertex of polygon");
   if (npoly==2) {add_linesegment(POLY[0], POLY[1]); continue;}
   Pnr = abs(POLY[0]); Qnr = abs(POLY[1]);
   for (i=2; i<npoly; i++)
   {  orient = orienta(Pnr, Qnr, abs(POLY[i]));
      if (orient != 0) break; // Normally, i = 2
   }
   if (orient < 0) continue;  // Backface
```

```
    for (i=1; i<=npoly; i++)
    {  ii = i % npoly; code = POLY[ii];
       vertexnr = abs(code);
       if (code<0) POLY[ii] = vertexnr; else
       add_linesegment(POLY[i-1], vertexnr);
    }
    // Division of a polygon into triangles:
    code = triangul(POLY, npoly, nrs_tr, orienta);
    if (code == -2) memproblem();
    if (code > 0)
    {  complete_triangles(code, ntr, nrs_tr);
       ntr += code;
    }
}
inpfil.close(); delete[] POLY; delete[] nrs_tr;
setupscreenlists(triangles, ntr);
trset = new[maxntrset];
if (!trset) memproblem();

// Draw all line segments as far as they are visible:
for (Pnr=minvertnr; Pnr<=maxvertnr; Pnr++)
{  ptr = V[Pnr].connect;
   if (ptr == &dummy || ptr == NULL) continue;
   // &dummy: Pnr not in use; NULL: no line segments stored
   Ps = V[Pnr].VT;
   for (iconnect=1; iconnect<=*ptr; iconnect++)
   {  Qnr = ptr[iconnect]; Qs = V[Qnr].VT;
      // Using the screen lists, we shall build the
      // set of triangles that may hide points of PQ:
      if (Ps.X < Qs.X || (Ps.X == Qs.X && Ps.Y < Qs.Y))
           {Left = Ps; Right = Qs;}
      else {Left = Qs; Right = Ps;}
      ileft  = colnr(Left.X); iright = colnr(Right.X);
      if (ileft != iright)
      {  deltay = Right.Y - Left.Y;
         deltax = Right.X - Left.X;
      }
      jbot = jtop = rownr(Left.Y);
      for (I=ileft; I<=iright; I++)
      {  jI = (I == iright ? rownr(Right.Y) :
         rownr(Left.Y +
         (Xcoord(I + 1) - Left.X) * deltay / deltax));
         LOWER[I] = min2(jbot,jI); jbot = jI;
         UPPER[I] = max2(jtop,jI); jtop = jI;
      }
      ntrset = 0;
```

```
        for (I=ileft; I<=iright; I++)
        for (J=LOWER[I]; J<=UPPER[I]; J++)
        {  pnode = SCREEN[I][J];
           while (pnode != NULL)
           {  trnr= pnode->jtr;
              /* Triangle trnr will be stored only if it is
                 not yet present in array trset (triangle set)
              */
              trset[ntrset] = trnr; /* sentinel */
              jtr = 0;
              while (trset[jtr] != trnr) jtr++;
              if (jtr == ntrset)
              {  ntrset++; // This means that trnr is stored
                 if (ntrset == maxntrset)
                 {  int *p = trset;
                    trset = new[maxntrset += 200];
                    if (!trset) memproblem();
                    for (int i=0; i<ntrset; i++) trset[i] = p[i];
                    delete[] p;
                 }
              }
              pnode = pnode->next;
           }
        }
        // Now trset[0],..., trset[ntrset-1] is the set of
        // triangles that may hide points of PQ.
        linesegment(point(Ps, V[Pnr].z, Pnr),
                    point(Qs, V[Qnr].z, Qnr), ntrset);
        dealwithlinkedstack();
     }
  }
  t2 = clock();
  endgr();
  cout << "Number of vertices:  " << vertexcount << endl;
  cout << "Number of triangles: " << ntr << endl;
  cout << "Time: " << int((t2-t1)/CLK_TCK + .5) << " s.\n";
  return 0;
}
```

B

Program Text HIDEFACE

```
/* HIDEFACE: A program for hidden-surface elimination
      To be linked together with
         GRSYS (see Appendix C),
         D3 (see Section 5.3),
         TRIANGUL (see Section 3.5), and
         FILL (see Section 4.4).
*/

#include <fstream.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <time.h>
#include <limits.h>
#include "grsys.h"

#include "d3.h"
#include "triangul.h"
#include "fill.h"

const int LARGE=32000; // Not greater than sqrt(LONG_MAX)
const double PIdiv180=0.0174532, BIG=1.e30;
double f, zfactor, xsmin, xsmax, ysmin, ysmax,
   rcolormin=BIG, rcolormax=-BIG, delta, zemin, zemax,
   xsC, ysC, XLCreal, YLCreal;
```

```
vec3 *vt, lightvector;

int k, hk, XLC, YLC, dummy, vertexcount;

struct vec_int {int X, Y;};
struct vertex{int X, Y, Z; int *connect;} *V;
struct triadata {vec3 normal; float h; int color;};
struct tria {int Anr, Bnr, Cnr; int Z; triadata *ptria;}
   *triangles, *ptriangle;

struct trianode{int trnr; trianode *next;};

inline int XLarge(double xs) {return int(XLCreal + f * (xs - xsC));}
inline int YLarge(double ys) {return int(YLCreal + f * (ys - ysC));}
inline int ZLarge(double ze) {return int((ze-zemin)*zfactor + 0.5);}

inline double xScreen(int X) {return xsC + (X - XLC)/f;}
inline double yScreen(int Y) {return ysC + (Y - YLC)/f;}
inline double zEye(int Z) {return Z/zfactor + zemin;}

inline int max2(int i, int j) {return i > j ? i : j;}
inline int min2(int i, int j) {return i < j ? i : j;}

inline int max3(int i, int j, int k) {return max2(i, max2(j, k));}
inline int min3(int i, int j, int k) {return min2(i, min2(j, k));}

void memproblem()
{  errmess("Not enough memory");
}

void skip(ifstream &inpfil) // Skip input characters until '\n'
{  char ch;
   inpfil.clear();
   do inpfil.get(ch); while (ch != '\n' && !inpfil.eof());
}

int orientation(int u1, int u2, int v1, int v2)
{  long det = long(u1) * v2 - long(u2) * v1;
   return det < -250 ? -1 : det > 250;
}

inline int orienta(int Pnr, int Qnr, int Rnr)
{  return orientation(V[Qnr].X - V[Pnr].X, V[Qnr].Y - V[Pnr].Y,
                      V[Rnr].X - V[Pnr].X, V[Rnr].Y - V[Pnr].Y);
}
```

```
void complete_triangles(int n, int offset, trianrs *nrs_tr)
// Complete triangles[offset],..., triangles[offset+n-1].
// Vertex numbers: nrs_tr[0],..., nrs_tr[n-1].
// These triangles belong to the same polygon. The equation
// of their plane is nx . x + ny . y + nz . z = h.
{  int i, Anr, Bnr, Cnr, ZA, ZB, ZC;
   unsigned Zmin, Zmax;
   double nx, ny, nz, ux, uy, uz, vx, vy, vz, factor, h,
      Ax, Ay, Az, Bx, By, Bz, Cx, Cy, Cz;
   tria *p;
   triadata *q = new triadata;
   if (q == NULL) memproblem();
   // If the polygon is an approximation of a circle, the
   // first three vertices may lie almost on the same line,
   // hence n/2 instead of 0 in the following for-statement:
   for (i = n/2; i<n; i++)
   {  Anr = nrs_tr[i].A; Bnr = nrs_tr[i].B; Cnr = nrs_tr[i].C;
      if (orienta(Anr, Bnr, Cnr) > 0) break;
   }
   ZA = V[Anr].Z; ZB = V[Bnr].Z; ZC = V[Cnr].Z;
   Az = zEye(ZA); Bz = zEye(ZB); Cz = zEye(ZC);
   Ax = xScreen(V[Anr].X) * Az; Ay = yScreen(V[Anr].Y) * Az;
   Bx = xScreen(V[Bnr].X) * Bz; By = yScreen(V[Bnr].Y) * Bz;
   Cx = xScreen(V[Cnr].X) * Cz; Cy = yScreen(V[Cnr].Y) * Cz;
   // Computation of vector product normvec = (B-A) x (C-A):
   ux = Bx - Ax; uy = By - Ay; uz = Bz - Az;
   vx = Cx - Ax; vy = Cy - Ay; vz = Cz - Az;
   nx = uy * vz - uz * vy;
   ny = uz * vx - ux * vz;
   nz = ux * vy - uy * vx;
   h = nx * Ax + ny * Ay + nz * Az;
   factor = 1/sqrt(nx * nx + ny * ny + nz * nz);
   q->normal.x = nx * factor;
   q->normal.y = ny * factor;
   q->normal.z = nz * factor;
   q->h = h * factor;
   for (i=0; i<n; i++)
   {  p = triangles + offset + i;
      p->Anr = nrs_tr[i].A;
      p->Bnr = nrs_tr[i].B;
      p->Cnr = nrs_tr[i].C;
      p->ptria = q;
      // Take the triangle side that has the greatest Z-range;
      // p->Z will be based on the midpoint of that side:
      Zmin = Zmax = V[p->Anr].Z;
      ZB = V[p->Bnr].Z; ZC = V[p->Cnr].Z;
```

```
      if (ZB < Zmin) Zmin = ZB; else if (ZB > Zmax) Zmax = ZB;
      if (ZC < Zmin) Zmin = ZC; else if (ZC > Zmax) Zmax = ZC;
      p->Z = (Zmin + Zmax)/2;
   }
}

extern "C" int comparetriangles(const void *p1, const void *p2)
{  return ((tria*)p1)->Z < ((tria*)p2)->Z ? -1 : 1;
}

void findrange(int i)
{  vec3 normal = triangles[i].ptria->normal;
   float rcolor;
   rcolor = dotproduct(normal, lightvector);
   if (rcolor < rcolormin) rcolormin = rcolor;
   if (rcolor > rcolormax) rcolormax = rcolor;
}

inline int to_pix(unsigned X)  // Rounding and converting
{  return (X + hk)/k;
}

void set_tr_color(int i)
{  vec3 normal = triangles[i].ptria->normal;
   int color;
   float rcolor;
   rcolor = dotproduct(normal, lightvector);
   color = 1 + (rcolor - rcolormin) * delta;
   if (color < 0) errmess("Negative color code");
   if (color >= ncolors) errmess("Color code too large");
         // (in case of a programming error)
   triangles[i].ptria->color = color;
}

void fill_window(int i, int Xmin, int Xmax, int Ymin, int Ymax)
{  // Fill entire window with color determined by normal vector
   // of triangle i
   int y;
   set_color(triangles[i].ptria->color);
   Xmin /= k; Xmax /= k;
   Ymin /= k; Ymax /= k;
   for (y=Ymin; y<=Ymax; y++) horline(Xmin, Xmax, y);
}

void fill_triangle(int i)
// Fill triangle i
```

```
{  tria *p = triangles + i;
   int X[3], Y[3],
      Anr = p->Anr, Bnr = p->Bnr, Cnr = p->Cnr;
   X[0] = to_pix(V[Anr].X);
   Y[0] = to_pix(V[Anr].Y);
   X[1] = to_pix(V[Bnr].X);
   Y[1] = to_pix(V[Bnr].Y);
   X[2] = to_pix(V[Cnr].X);
   Y[2] = to_pix(V[Cnr].Y);
   set_color(p->ptria->color);
   fill(X, Y, 3);  // See Section 4.4
}

int inside_triangle(int X, int Y,
   int XA, int YA, int XB, int YB, int XC, int YC)
{  // Does (X, Y) lie on or inside triangle ABC?
   return orientation(XB-XA, YB-YA, X-XA, Y-YA) >= 0 &&
          orientation(XC-XB, YC-YB, X-XB, Y-YB) >= 0 &&
          orientation(XA-XC, YA-YC, X-XC, Y-YC) >= 0;
}

void deletelist(trianode *start)
{  trianode *p;
   while (start != NULL)
   {  p = start;
      start = start->next;
      delete p;
   }
}

inline void swap(int &x, int &y)
{  int t=x;
   x = y; y = t;
}

int intersectvertical(vec_int &A, vec_int &B, int X,
   int Ymin, int Ymax)
// Does line segment AB have points in common with the
// vertical line segment {(X, Ymin), (X, Ymax)}?
{  int XA=A.X, YA=A.Y, XB=B.X, YB=B.Y;
   long dX, dY, YdX;
   if (XA < X && XB < X || XA > X && XB > X) return 0;
   if (XA == X && YA >= Ymin && YA <= Ymax ||
       XB == X && YB >= Ymin && YB <= Ymax) return 1;
   if (XA == XB)
      return XA == X &&
```

```
      (long(YA - Ymax) * (YB - Ymax) < 0 ||
       long(YA - Ymin) * (YB - Ymin) < 0);
   if (XA > XB) {swap(XA, XB); swap(YA, YB);}
   dX = XB - XA; dY = YB - YA;
   YdX = YA * dX + (X - XA) * dY;
   return YdX >= Ymin * dX  &&  YdX <= Ymax * dX;
}

int intersecthorizontal(vec_int &A, vec_int &B, int Y,
   int Xmin, int Xmax)
// Does line segment AB have points in common with the
// horizontal line segment {(Xmin, Y), (Xmax, Y)}?
{  int XA=A.X, YA=A.Y, XB=B.X, YB=B.Y;
   long dX, dY, XdY;
   if (YA < Y && YB < Y || YA > Y && YB > Y) return 0;
   if (YA == Y && XA >= Xmin && XA <= Xmax ||
       YB == Y && XB >= Xmin && XB <= Xmax) return 1;
   if (YA == YB)
      return YA == Y &&
      (long(XA - Xmax) * (XB - Xmax) < 0 ||
       long(XA - Xmin) * (XB - Xmin) < 0);
   if (YA > YB) {swap(XA, XB); swap(YA, YB);}
   dX = XB - XA; dY = YB - YA;
   XdY = XA * dY + (Y - YA) * dX;
   return XdY >= Xmin * dY  &&  XdY <= Xmax * dY;
}

int overlap(unsigned Xmin, unsigned Xmax,
            unsigned Ymin, unsigned Ymax,
            vec_int &A, vec_int &B, vec_int &C)
// Does triangle ABC have points in common with window?
// (We know that there is span overlap.)
{  if (Xmin == Xmax && Ymin == Ymax)
   return inside_triangle(Xmin, Ymin, A.X, A.Y, B.X, B.Y, C.X, C.Y);
   int X = (Xmin + Xmax)/2, Y = (Ymin + Ymax)/2;
   // Here is a very long return-statement:
   return
   // Does triangle ABC lie inside window?
   (A.X >= Xmin && A.X <= Xmax && A.Y >= Ymin && A.Y <= Ymax ||
    B.X >= Xmin && B.X <= Xmax && B.Y >= Ymin && B.Y <= Ymax ||
    C.X >= Xmin && C.X <= Xmax && C.Y >= Ymin && C.Y <= Ymax) ||

   // Does window center lie inside (or on) triangle ABC?
   inside_triangle(X, Y, A.X, A.Y, B.X, B.Y, C.X, C.Y) ||

   // Does a triangle side and a window edge intersect?
```

```
   intersectvertical(A, B, Xmin, Ymin, Ymax) ||   // AB, left
   intersectvertical(A, B, Xmax, Ymin, Ymax) ||   // AB, right
   intersecthorizontal(A, B, Ymin, Xmin, Xmax) || // AB, lower
   intersecthorizontal(A, B, Ymax, Xmin, Xmax) || // AB, upper
   intersectvertical(B, C, Xmin, Ymin, Ymax) ||   // BC, left
   intersectvertical(B, C, Xmax, Ymin, Ymax) ||   // BC, right
   intersecthorizontal(B, C, Ymin, Xmin, Xmax) || // BC, lower
   intersecthorizontal(B, C, Ymax, Xmin, Xmax) || // BC, upper
   intersectvertical(C, A, Xmin, Ymin, Ymax) ||   // CA, left
   intersectvertical(C, A, Xmax, Ymin, Ymax) ||   // CA, right
   intersecthorizontal(C, A, Ymin, Xmin, Xmax) || // CA, lower
   intersecthorizontal(C, A, Ymax, Xmin, Xmax);   // CA, upper
}

double distance(int itria, int X, int Y)
// Consider the line through viewpoint E and point (X, Y) of the
// screen. We are interested in the point where this line intersects
// triangle itria. The ze coordinate of this point will be returned.
{  static double dist0=0;
   static int X0=0, Y0=0;
   static triadata *ptria0=NULL;
   // Static variables in case the same point (X, Y) is used for two
   // successive triangles belonging to the same polygon (and
   // therefore having the same ptria pointer).
   double a, b, c, h, xs, ys;
   tria* triaptr = triangles + itria; // = &triangles[itria]
   triadata *ptria=triaptr->ptria;
   if (ptria != ptria0 || X != X0 || Y != Y0)
   {  a = triaptr->ptria->normal.x;
      b = triaptr->ptria->normal.y;
      c = triaptr->ptria->normal.z;
      h = triaptr->ptria->h;
      xs = xScreen(X); ys = yScreen(Y);
      dist0 = h * sqrt(xs*xs + ys*ys + 1)/(a*xs + b*ys + c);
      X0 = X; Y0 = Y; ptria0 = ptria;
   }
   return dist0;
}

void add_to_list(trianode **pstart, trianode **pend, int trnr)
{  trianode *p1;
   p1 = new(trianode);
   if (p1 == NULL) memproblem();
   if (*pstart == NULL) *pstart = *pend = p1; else
   {  (*pend)->next = p1;
       *pend = p1;
```

```
   }
   p1->trnr = trnr; p1->next = NULL;
}

void Warnock(unsigned Xmin, unsigned Xmax,
             unsigned Ymin, unsigned Ymax, trianode *start)
{  int Anr, Bnr, Cnr, trnr,
      Xmid=(Xmin+Xmax)/2, Ymid=(Ymin+Ymax)/2,
      win2, win3, win4, count,
      XmidA, XmidB, YmidA, YmidB,
      XA, YA, XB, YB, XC, YC,
      Xtrmin, Xtrmax, Ytrmin, Ytrmax,
      wide = Xmax > Xmin, // Window more than one pixel wide?
      high = Ymax > Ymin, // Window more than one pixel high?
      more_than_one_triangle = (start->next != NULL);
   double dist, distmin;
   trianode *p, *p1, *p2, *p3, *p4, *p1end, *p2end,
      *p3end, *p4end, *paux, *pmin;
   vec_int A, B, C;
   tria *triaptr;
   distmin = 1e30; pmin = start;

   // Do all triangles of the linked list surround the window?
   for (p=start; p; p = p->next)
   {  trnr = p->trnr;
      triaptr = triangles+trnr;
      Anr = triaptr->Anr; XA = V[Anr].X; YA = V[Anr].Y;
      Bnr = triaptr->Bnr; XB = V[Bnr].X; YB = V[Bnr].Y;
      Cnr = triaptr->Cnr; XC = V[Cnr].X; YC = V[Cnr].Y;

      if (!inside_triangle(Xmin, Ymin, XA, YA, XB, YB, XC, YC) ||
          (wide &&
          !inside_triangle(Xmax, Ymin, XA, YA, XB, YB, XC, YC)) ||
          (wide && high &&
          !inside_triangle(Xmax, Ymax, XA, YA, XB, YB, XC, YC)) ||
          (high &&
          !inside_triangle(Xmin, Ymax, XA, YA, XB, YB, XC, YC)))
         break;

      // Triangle *p surrounds the window:
      if (more_than_one_triangle)
      {  dist = distance(trnr, Xmid, Ymid); count = 1;
         if (!wide && !high)
         // Use also four neigboring points,
         // as far as these lie inside triangle ABC:
         {  if (inside_triangle(
```

```
                Xmid - hk, Ymid - hk, XA, YA, XB, YB, XC, YC))
                dist += distance(trnr, Xmid-hk, Ymid-hk), count++;
            if (inside_triangle(
                Xmid + hk, Ymid + hk, XA, YA, XB, YB, XC, YC))
                dist += distance(trnr, Xmid+hk, Ymid+hk), count++;
            if (inside_triangle(
                Xmid + hk, Ymid - hk, XA, YA, XB, YB, XC, YC))
                dist += distance(trnr, Xmid+hk, Ymid-hk), count++;
            if (inside_triangle(
                Xmid - hk, Ymid + hk, XA, YA, XB, YB, XC, YC))
                dist += distance(trnr, Xmid-hk, Ymid+hk), count++;
            dist /= count;
        }
        if (dist < distmin) {distmin = dist; pmin = p;}
    }
}

// If now p == NULL, no break-statement was executed,
// so all triangles surround window, and pmin points to the
// closest triangle.
if (p == NULL)
{   fill_window(pmin->trnr, Xmin, Xmax, Ymin, Ymax);
    deletelist(start);
    return;
}
if (!wide && !high) // No further division possible
{   fill_window(start->trnr, Xmin, Xmax, Ymin, Ymax);
    deletelist(start);
    return;
}

// Divide window into four smaller ones:
p1 = p2 = p3 = p4 = p1end = p2end = p3end = p4end = NULL;
XmidA = Xmid/k * k; XmidB = XmidA + k;
YmidA = Ymid/k * k; YmidB = YmidA + k;
win2 = XmidB <= Xmax;
win4 = YmidB <= Ymax;
win3 = win2 && win4;

for (p=start; p;  )
{   trnr = p->trnr;
    triaptr = triangles+trnr;
    Anr = triaptr->Anr; A.X = V[Anr].X; A.Y = V[Anr].Y;
    Bnr = triaptr->Bnr; B.X = V[Bnr].X; B.Y = V[Bnr].Y;
    Cnr = triaptr->Cnr; C.X = V[Cnr].X; C.Y = V[Cnr].Y;
    Xtrmin = min3(A.X, B.X, C.X);
```

```
      Xtrmax = max3(A.X, B.X, C.X);
      Ytrmin = min3(A.Y, B.Y, C.Y);
      Ytrmax = max3(A.Y, B.Y, C.Y);
      if (XmidA >= Xtrmin && Xmin <= Xtrmax &&
          YmidA >= Ytrmin && Ymin <= Ytrmax &&
          overlap(Xmin, XmidA, Ymin, YmidA, A, B, C))
         add_to_list(&p1, &p1end, trnr);
      if (win2 && Xmax >= Xtrmin && XmidB <= Xtrmax &&
          YmidA >= Ytrmin && Ymin <= Ytrmax &&
          overlap(XmidB, Xmax, Ymin, YmidA, A, B, C))
         add_to_list(&p2, &p2end, trnr);
      if (win3 && Xmax >= Xtrmin && XmidB <= Xtrmax &&
          Ymax >= Ytrmin && YmidB <= Ytrmax &&
          overlap(XmidB, Xmax, YmidB, Ymax, A, B, C))
         add_to_list(&p3, &p3end, trnr);
      if (win4 && XmidA >= Xtrmin && Xmin <= Xtrmax &&
          Ymax >= Ytrmin && YmidB <= Ytrmax &&
          overlap(Xmin, XmidA, YmidB, Ymax, A, B, C))
         add_to_list(&p4, &p4end, trnr);
      paux = p; p = p->next; delete(paux);
   }
   if (p4) Warnock(Xmin, XmidA, YmidB, Ymax, p4);
   if (p3) Warnock(XmidB, Xmax, YmidB, Ymax, p3);
   if (p1) Warnock(Xmin, XmidA, Ymin, YmidA, p1);
   if (p2) Warnock(XmidB, Xmax, Ymin, YmidA, p2);
}

int main(int argc, char *argv[])
{  int i, ii, vertexnr, maxvertnr=0, maxnpoly, totnrtria,
      code, ntr=0, *POLY, npoly, k1, k2, XLmax, YLmax,
      nvertex, Pnr, Qnr, orient, testtria[3];
   trianrs *nrs_tr; // Type trianrs is declared in triangul.h
   clock_t t1, t2;
   trianode *startptr, *endptr;
   double fx, fy, rho, theta, phi, xs, ys, xe, ye, ze,
      xsrange, ysrange, xmin=BIG, xmax=-BIG,
      ymin=BIG, ymax=-BIG, zmin=BIG, zmax=-BIG, x, y, z;
   char ch, method;
   ifstream inpfil;
   vec3 ve, Objpoint, Pnew;
   if (argc < 2 || (inpfil.open(argv[1]), !inpfil))
      errmess("Input file not correctly specified.");
   cout << "Please wait...\n";

   for ( ; ; )
   {  inpfil >> i >> x >> y >> z;
```

```
    if (i % 50 == 0)
    cout << "."; // Show that something is happening...
    if (i > maxvertnr) maxvertnr = i;
    if (x < xmin) xmin = x;
    if (x > xmax) xmax = x;
    if (y < ymin) ymin = y;
    if (y > ymax) ymax = y;
    if (z < zmin) zmin = z;
    if (z > zmax) zmax = z;
    if (inpfil.fail() || inpfil.eof()) break;
}

cout << endl;
inpfil.close();
inpfil.open(argv[1]);
nvertex = maxvertnr + 1;
vt = new vec3[nvertex];  // Preliminary vertex list
if (!vt) errmess("Too many vertices");
Objpoint = vec3(.5*(xmin+xmax), .5*(ymin+ymax),
                .5*(zmin+zmax));
cout << "x range: " << xmin << " ... " << xmax << endl;
cout << "y range: " << ymin << " ... " << ymax << endl;
cout << "z range: " << zmin << " ... " << zmax << endl;
cout << "Center: " << Objpoint.x << " " << Objpoint.y
     << " " << Objpoint.z << endl;
cout << "Use center as default central object point? (Y/N): ";
cin >> ch; ch = toupper(ch);
if (ch == 'N')
{  cout << "Enter coordinates of central object point:\n";
   cin >> Objpoint;
}
rho = xmax - xmin;
if (ymax - ymin > rho) rho = ymax - ymin;
if (zmax - zmin > rho) rho = zmax - zmin;
rho *= 3; theta = 20; phi = 70;
cout << "Use default viewpoint (rho = " << rho
     << ", theta = 20, phi = 70)? (Y/N): ";
cin >> ch; ch = toupper(ch);
if (ch == 'N')
{  cout << "Enter spherical coordinates rho, theta, phi of\n";
   cout << "viewpoint E (phi = angle between z-axis and OE)\n";
   cin >> rho >> theta >> phi;
}
cout << "Use default light vector 1 -1 0 ? (Y/N): ";
cin >> ch; ch = toupper(ch);
lightvector = vec3(1, -1, 0);
```

```
if (ch == 'N')
{  cout << "Enter\n"
           "component from left to right,\n"
           "component from bottom to top, and\n"
           "component from front to back\n"
   "of light vector (only their ratio matters): ";
   cin >> lightvector;
}
cout << "\nYou must choose between the following algorithms:\n"
     "(P) a fast Painter's algorithm, which may "
     "give incorrect results, and\n"
     "(W) Warnock's algorithm, which is more "
     "accurate but slower.\n";
do
{  cout << "Your choice? (P/W): ";
   cin >> method; method = toupper(method);
}  while (method != 'P' && method != 'W');
t1 = clock(); coeff(rho, theta*PIdiv180, phi*PIdiv180);
for (i=0; i<nvertex; i++) vt[i].z = -1e6; // Not in use
// Read vertices:
xsmin=ysmin=zemin=BIG; xsmax=ysmax=zemax=-BIG;
cout << "Please wait...\n";

for ( ; ; )
{  inpfil >> i >> x >> y >> z;
   if (i % 50 == 0)
   cout << "."; // Show that something is happening...
   if (inpfil.fail() || inpfil.eof()) break;
   vertexcount++;
   if (i < 0 || i >= nvertex)
   errmess("Too many vertices or illegal vertex number");
   Pnew.x = x - Objpoint.x;
   Pnew.y = y - Objpoint.y;
   Pnew.z = z - Objpoint.z;
   eyecoord(Pnew, ve);
   xe = ve.x; ye = ve.y; ze = ve.z;
   if (ze < 0)
   errmess("\nObject point O and a vertex on different "
           "sides of viewpoint E."
           "Try a larger value for rho.\n");
   xs = xe/ze;  ys = ye/ze;
   if (xs < xsmin) xsmin = xs; if (xs > xsmax) xsmax = xs;
   if (ys < ysmin) ysmin = ys; if (ys > ysmax) ysmax = ys;
   if (ze < zemin) zemin = ze; if (ze > zemax) zemax = ze;
   vt[i] = ve;
}
```

```
cout << endl;
if (xsmin == BIG) errmess("Incorrect input file");
initgr();
shaded_colors();
// Compute screen constants:
xsrange = xsmax - xsmin; ysrange = ysmax - ysmin;
xsC = 0.5 * (xsmin + xsmax);
ysC = 0.5 * (ysmin + ysmax);
k1 = LARGE/(X__max+1); k2 = LARGE/(Y__max+1);
k = min2(k1, k2); hk = k/2;  // k = 50, hk = 25 with VGA
XLmax = k * (X__max+1); YLmax = k * (Y__max+1);
// Pixel coordinates: Xpix = to_pix(X) and Ypix = to_pix(Y)
XLC = XLmax/2; YLC = YLmax/2;
XLCreal = XLC + 0.5; YLCreal = YLC + 0.5;
fx = XLmax/xsrange; fy = YLmax/ysrange;
f = 0.95 * (fx < fy ? fx : fy);
zfactor = LARGE/(zemax - zemin);
V = new vertex[nvertex];
if (V == NULL) memproblem();

// Initialize vertex array:
for (i=0; i<nvertex; i++)
{  if (vt[i].z < -1e5)
   {V[i].connect = &dummy; continue;} // V[i] not in use
   V[i].connect = NULL;
   xs = vt[i].x / vt[i].z;
   ys = vt[i].y / vt[i].z;
   V[i].X = XLarge(xs);
   V[i].Y = YLarge(ys);
   V[i].Z = ZLarge(vt[i].z);
}
delete[] vt;  // Replaced with V

// Find the maximum number of vertices in one polygon
// and the total number of non-backface triangles:
skip(inpfil); // Skip the word 'Faces'
maxnpoly = totnrtria = 0;
for ( ; ; )
{  for (npoly=0; ; npoly++)
   {  inpfil >> i; i = abs(i);
      if (inpfil.fail() || inpfil.eof()) break;
      if (i >= nvertex || V[i].connect == &dummy)
         errmess("Undefined vertex number used.");
      if (npoly < 3) testtria[npoly] = i;
   }
   if (inpfil.eof()) break;
```

```
   if (npoly > maxnpoly) maxnpoly = npoly;
   skip(inpfil);
   if (npoly < 3) continue; // Ignore 'loose' line segment
   if (orienta(testtria[0], testtria[1], testtria[2]) >= 0)
      totnrtria += npoly - 2;
}

inpfil.close(); inpfil.open(argv[1]); // Rewind
do inpfil >> x; while (!inpfil.fail() && !inpfil.eof());

triangles = new tria[totnrtria];
POLY = new int[maxnpoly];
nrs_tr = new trianrs[maxnpoly-2];

if (!triangles || !POLY || !nrs_tr) memproblem();

// Read object faces and store triangles:
skip(inpfil); // Skip the word 'Faces'
for ( ; ; )
{  npoly = 0;
   for ( ; ; )
   {  inpfil >> i; i = abs(i);
      if (inpfil.fail() || inpfil.eof()) break;
      if (npoly == maxnpoly)
         errmess("Programming error: maxnpoly");
      POLY[npoly++] = i;
   }
   if (inpfil.eof()) break;
   skip(inpfil);
   if (npoly < 3) continue; // Ignore 'loose' line segment
   Pnr = abs(POLY[0]); Qnr = abs(POLY[1]);
   for (i=2; i<npoly; i++)
   {  orient = orienta(Pnr, Qnr, abs(POLY[i]));
      if (orient != 0) break; // Normally, i = 2
   }
   if (orient < 0) continue; // Backface

   for (i=1; i<=npoly; i++)
   {  ii = i % npoly; vertexnr = abs(POLY[ii]);
      if (V[vertexnr].connect == &dummy)
         errmess("Undefined vertex number used");
   }
   // Division of a polygon into triangles:
   code = triangul(POLY, npoly, nrs_tr, orienta);
   if (code == -2) memproblem();
   if (code > 0)
```

```
      {  if (ntr + code > totnrtria)
            errmess("Programming error: totnrtria");
         complete_triangles(code, ntr, nrs_tr);
         ntr += code;
      }
   }

   inpfil.close();
   delete[] POLY; delete[] nrs_tr;

   for (i=ntr-1; i>=0; i--) findrange(i);
   delta = 0.999 * (ncolors - 1)/(rcolormax - rcolormin + 0.001);
   for (i=ntr-1; i>=0; i--) set_tr_color(i);

   if (method == 'P')  // Painter's algorithm:
   {  qsort(triangles, ntr, sizeof(tria), comparetriangles);
      // triangles[0] is the nearest triangle
      for (i=ntr-1; i>=0; i--) fill_triangle(i);
   }  else             // Warnock's algorithm:
   {  startptr = endptr = NULL;
      for (i=0; i<ntr; i++) add_to_list(&startptr, &endptr, i);
      Warnock(0, XLmax-k, 0, YLmax-k, startptr);
   }

   t2 = clock();
   endgr();
   if (method == 'P')
   {  cout << "If the results were not correct, run the "
              "program\n";
      cout << "once again, using Warnock's algorithm.\n";
   }
   cout << "Number of vertices:  " << vertexcount << endl;
   cout << "Number of triangles: " << ntr << endl;
   cout << "Time: " << int((t2-t1)/CLK_TCK + .5) << " s.\n";
   return 0;
}
```

C

Program Text GRSYS

(based on Borland C++ and VGA)

For example, enter the following line to compile and link HIDELINE:

```
bcc -mh hideline.cpp grsys.cpp triangul.cpp d3.cpp graphics.lib
```

```
// GRSYS.H: Graphics primitives (Header file)
extern float x_min, x_max, y_min, y_max,          // Section 1.2
   x_center, y_center, r_max,                     // Section 1.2
   density;                                       // Section 4.2
extern int ingraphicsmode, // 1: graphics mode; 0: text mode
   X__max, Y__max,                                // Section 4.1
   ncolors, foregrcolor, backgrcolor;             // Section 4.1
void initgr(char *hpgfile=0);                     // Section 1.2
void endgr();                                     // Section 1.2
void to_text();               // Immediately back to text mode
void move(float x, float y);                      // Section 1.2
void draw(float x, float y);                      // Section 1.2
void errmess(char *s);                            // Section 2.6
int get_maxcolor(void);                           // Section 4.1
void set_color(int color);                        // Section 4.1
void set_backgroundcolor(int color);              // Section 4.1
void putpix(int X, int Y);                        // Section 4.1
void horline(int xleft, int xright, int y);       // Section 4.2
void draw_line(int X1, int Y1, int X2, int Y2); // Section 4.2
int ix(float x), iy(float y);              // Sections 4.1, 4.2
void set_rgb_palette(int colornr, int R, int G, int B); // 7.1
void shaded_colors(void);                         // Section 7.1
```

```
// GRSYS.CPP: Graphics primitives
//            (based on Borland C++ and VGA)
#include <fstream.h>
#include <stdlib.h>
#include <graphics.h>
#include <conio.h>
#include <dos.h>
#include "grsys.h"
const float BIG = 1e30;
int X__max, Y__max, foregrcolor, backgrcolor, ncolors,
  ingraphicsmode=0;
float x_min=0, y_min=0, x_max=10, y_max,
   x_center, y_center, r_max, density;

static int outside=0;
ofstream fplot;
static int hpgoutput=0;

static int grbrfun(void)
// Used by  ctrlbrk, to specify what to do with a console break
{  to_text(); return 0;       // Return to text mode before exit
}

static int txtbrfun(void)
{  return 0;
}

void set_color(int color)
{  setcolor(color);
   foregrcolor = color;
}

void set_backgroundcolor(int color)
{  setbkcolor(color);
   backgrcolor = color;
}

int get_maxcolor(void)
{  return getmaxcolor();
}

inline int plotcoor(float x) {return int(1000 * (x) + 0.5);}

void set_rgb_palette(int colornr, int R, int G, int B)
{  palettetype pal;
   getpalette(&pal);
```

```
   setrgbpalette(pal.colors[colornr], R, G, B);
}
void shaded_colors(void)
{  int i, j, red, green, blue;
   if (ncolors != 16) errmess("No standard VGA in use");
   ifstream palfile("palette.txt");
   if (palfile) // Is there a palette file?
   {  // If so, use it
      for (i=0; i<ncolors; i++)
      {  palfile >> red >> green >> blue;
         if (palfile.fail()) errmess("Incorrect file PALETTE.TXT");
         set_rgb_palette(i, red, green, blue);
      }
   }  else // Use blue background and yellow foreground
   {  set_rgb_palette(0, 0, 0, 63);  // 0 = dark blue
      for (i=1; i<ncolors; i++)
      {  j = 4 * i + 3;
         set_rgb_palette(i, j, j, 0);
         // Shades of yellow: 1 = very dark, 15 = very bright
      }
   }
}

void initgr(char *hpgfile) // Default argument (see also GRSYS.H)
{  int gdriver=DETECT, gmode, w, h, errorcode;
   initgraph(&gdriver, &gmode, "c:\\borlandc\\bgi");
   // If not found in directory C:\BORLANDC\BGI, the file EGAVGA.BGI
   // is also searched for in the current directory.
   errorcode = graphresult();
   if (errorcode != grOk)
   {  cout << grapherrormsg(errorcode) << endl;
      exit(1);
   }
   ingraphicsmode = 1;
   ctrlbrk(grbrfun);   // Set break trap, Borland C++
   ncolors = getmaxcolor() + 1;
   set_rgb_palette(0, 0, 0, 63);          // Dark blue
   set_rgb_palette(14, 63, 63, 0);        // Yellow
   set_backgroundcolor(0); set_color(14); // Default (VGA)
   X__max = getmaxx(); Y__max = getmaxy();
   getaspectratio(&w, &h);
   if (w != h) errmess("No VGA in use (square pixels required)");
   density = X__max / (x_max - x_min);    // See Section 4.2
   y_max = y_min + Y__max/density;
   x_center = 0.5 * (x_min + x_max);
   y_center = 0.5 * (y_min + y_max);
```

```
   r_max = (x_center < y_center ? x_center : y_center);
   if (hpgfile)
   {  fplot.open(hpgfile);
      if (fplot) // that is, if plotfile can be opened:
      {  fplot << "IN;SP0;SC0,10000,0," << plotcoor(y_max-y_min)
               << ";\n";
         hpgoutput = 1;
      }
   }
}

void to_text(void)  // Revert to text mode
{  if (ingraphicsmode)
   { closegraph();
     ctrlbrk(txtbrfun);
     // Restore default break interrupt handler
     ingraphicsmode = 0;
     if (outside)
        cout << "There was an attempt to draw lines "
                "outside screen boundaries.\n";
   }
}

void endgr()
{  getch(); to_text();
}

static int IX(float x)
{  int X = int(density * (x - x_min));
   if (X < 0) {X = 0; outside = 1;}
   if (X > X__max) {X = X__max; outside = 1;}
   return X;
}

static int IY(float y)
{  int Y = Y__max - int(density * (y - y_min));
   if (Y < 0) {Y = 0; outside = 1;}
   if (Y > Y__max) {Y = Y__max; outside = 1;}
   return Y;
}

int ix(float x) {return IX(x);}
int iy(float y) {return Y__max - IY(y);}

void move(float x, float y)
{  moveto(IX(x), IY(y));
```

```
   if (hpgoutput)
      fplot << "PU;PA" << plotcoor(x - x_min) << ","
            << plotcoor(y-y_min) << ";\n";
}

void draw(float x, float y)
{  kbhit();       // Enable Ctrl-Break
   lineto(IX(x), IY(y));
   if (hpgoutput)
      fplot << "PD;PA" << plotcoor(x - x_min) << ","
            << plotcoor(y-y_min) << ";\n";
}

void putpix(int X, int Y){putpixel(X, Y__max - Y, foregrcolor);}

void draw_line(int X1, int Y1, int X2, int Y2)
{  kbhit();  // Enable Ctrl-Break
   line(X1, Y__max - Y1, X2, Y__max - Y2);
}

void horline(int xleft, int xright, int y)
{  kbhit();  // Enable Ctrl-Break
   line(xleft, Y__max-y, xright, Y__max-y);
}

void errmess(char *s)
{  if (ingraphicsmode) to_text();
   cout << s << endl;
   exit(1);
}

// To be used only by the Borland graphics functions themselves:

void far * far _graphgetmem(unsigned size)
{ char *p;
  p = new char[size];
  if (p == NULL) {cout << "Mem. space in _graphgetmem"; exit(1);}
  return p;
}

void far _graphfreemem(void far *ptr, unsigned size)
{ delete ptr;
}
```

Bibliography

Ammeraal, L. (1988) *Interactive 3D Computer Graphics*, Chichester: John Wiley.

Ammeraal, L. (1989) *Graphics Programming in Turbo C*, Chichester: John Wiley.

Ammeraal, L. (1991) *C++ for Programmers*, Chichester: John Wiley.

Ammeraal, L. (1992) *Programs and Data Structures in C, 2nd Edition*, Chichester: John Wiley.

Borland (1991) *Borland C++*, Scotts Valley, CA: Borland International.

Coxeter, H. S. M. (1961) *Introduction to Geometry*, New York: John Wiley.

Escher, M. C., *et al.* (1972) *The World of M. C. Escher*, New York: Harry N. Abrams.

Foley, J. D., A. van Dam, S. K. Feiner, and J. F. Hughes (1990) *Computer Graphics - Principles and Practice, 2nd Edition*, Reading, MA: Addison-Wesley.

Hearn, D., and M. P. Baker (1986) *Computer Graphics*, Englewood Cliffs, NJ: Prentice-Hall.

Kreyszig, E. (1962) *Advanced Engineering Mathematics*, New York: John Wiley.

Newman, M. N., and R. F. Sproull (1979) *Principles of Interactive Computer Graphics*, New York: McGraw-Hill.

Penna, M. A., and R. R. Patterson (1986) *Projective Geometry and its Applications to Computer Graphics*, Englewood Cliffs, NJ: Prentice-Hall.

Salmon, R., and M. Slater (1987) *Computer Graphics - Systems & Concepts*, Wokingham: Addison-Wesley.

Stroustrup, B. (1991) *The C++ Programming Language, 2nd Edition*, Reading, MA: Addison-Wesley.

Watt, A. (1989) *Fundamentals of Three-Dimensional Computer Graphics*, Wokingham: Addison-Wesley.

Index

WPC

PROGRAMMING PRINCIPLES IN COMPUTER GRAPHICS

Second Edition

Leendert Ammeraal

Hogeschool Utrecht, The Netherlands

For anyone who is interested in experimenting with computer graphics (or is teaching those who are) ... this book should get them hooked...

—*Computer-Aided Design* (review of the first edition)

In its second, updated edition, examples in this introduction to graphics programming have been rewritten in the C++ language.

The author uses a host of ready-to-run programs and worked examples to illuminate general principles and geometric techniques for the creation of both 2D and 3D graphical objects.

Still accessible to the C programmer, the book benefits from some elegant programming concepts of C++. It has been expanded to include subjects related to pixels, such as Bresenham's algorithms for lines and circles, polygon-filling and hidden-surface elimination; its approach is machine-independent. Matters of perspective are looked at in detail and the use of color is discussed and illustrated.

Spread throughout the text, numerous exercises encourage the reader to test and improve programming skills. A useful instructive tool for both student and teaching professional, this book will be a fine starting point for any graphics programmer.

ISBN 0-471-93128-4

9 780471 931287

JOHN WILEY & SONS
Chichester • New York • Brisbane • Toronto • Singapore

PROGRAMMING PRINCIPLES IN COMPUTER GRAPHICS *Second Edition*

Ammeraal

WILEY